Achieving High Performance in Parallel Applications via Kernel-Application Interaction

Robert W. Wisniewski

Technical Report 615 and Ph.D. Thesis
April 1996

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

Achieving High Performance in Parallel Applications via Kernel-Application Interaction

by

Robert W. Wisniewski

Submitted in Partial Fulfillment

of the

Requirements for the Degree

Doctor of Philosophy

Supervised by

Christopher M. Brown

Department of Computer Science
The College
Arts and Sciences

University of Rochester
Rochester, New York

1996

REPORT DOCUMENTATION PAGE

Form Approved

OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	**2. REPORT DATE** April 1996	**3. REPORT TYPE AND DATES COVERED** technical report and Ph.D. thesis

4. TITLE AND SUBTITLE

Achieving High Performance in Parallel Applications via Kernel-Application Interaction

5. FUNDING NUMBERS

ONR N00014-93-I-0221ONR

N00014-92-J-1801 / ARPA 8930

6. AUTHOR(S)

Robert W. Wisniewski

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESSES

Computer Science Dept.
734 Computer Studies Bldg.
University of Rochester
Rochester NY 14627-0226

8. PERFORMING ORGANIZATION

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESSES(ES)

Office of Naval Research	ARPA
Information Systems	3701 N. Fairfax Drive
Arlington VA 22217	Arlington VA 22203

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

TR 615

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Distribution of this document is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

(see title page)

14. SUBJECT TERMS

real-time; multiprogramming; synchronization; scalable synchronization; adaptive runtime environment; Ephor; SPARTA

15. NUMBER OF PAGES

156 pages

16. PRICE CODE

free to sponsors; else $7.00

17. SECURITY CLASSIFICATION OF REPORT unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** unclassified	**20. LIMITATION OF ABSTRACT** UL

Curriculum Vitae

Robert William Wisniewski [redacted]. He grew up in Lewiston, New York and graduated valedictorian of Niagara Wheatfield High School in 1986. He attended Cornell University where he worked as a computer operator and later as a supervisor for Cornell Computer Services. While at Cornell, he swam on the Varsity team for four years and was Captain and MVP in his senior year. He was a Cornell National Scholar from 1986 until he graduated in May of 1990. In September of 1990 he entered graduate school at the University of Rochester and received his Masters of Science in Computer Science in May of 1992. He served as a research assistant to Professor Christopher M. Brown working on high performance systems support for real-world applications and also worked closely with Professor Michael L. Scott and Leonidas Kontothanassis on multiprogrammed multiprocessor synchronization techniques. While at Rochester he was a teaching assistant for an honors introductory course and taught two lectures for "An Introduction to Computer Science". In 1994 he interned and later consulted for Silicon Graphics Incorporated; and in 1995 he interned at Digital Equipment Corporation's Cambridge Research Laboratory. In 1994, and again in 1995, he received an ARPA fellowship in High Performance Computing.

Acknowledgments

I owe immeasurable thanks to my mother and late father. Throughout my life they have provided guidance, support, and motivation. To my father, from whom I acquired curiosity and the desire to learn and explore, to my mother who taught me persistence, hard work, and dedication, as well as showing me the way to a PhD, to both for caring and supporting each other and our family through good and difficult, to them I say thank you; it is impossible to ever repay my gratitude, but know it's there.

Along the road to a PhD are many decisions, and undoubtably my best was picking Chris as an advisor. Chris had confidence in me early on before others; he provided tremendous guidance, and showed amazing patience in my early flounderings. And later, as I fortunately became more adept at the research game, Chris allowed me the freedom and flexibility to pursue all my interests – I could ask for no more.

Part of this thesis draws on work performed with one of my committee members, Professor Michael L. Scott, and fellow graduate student, Leonidas Kontothanassis. Together we examined and conquered the effect of multiprogramming on multiprocessor synchronization. Another part of this thesis is work on a real-world shepherding application involving vision and robot manipulation. For some of the implementation of this application I was fortunate to have the talents of Peter von Kaenel, a masters student. I also owe thanks for the helpful suggestions from my committee members. Michael Scott, in addition to working on synchronization, provided good insights into system interfaces and the importance of empirical numbers. Tom LeBlanc's past experience in real-time systems and insightful criticisms proved valuable, and Professor Abraham Seidmann's perspectives were always helpful. My intern experiences greatly broadened and strengthened my views. I owe thanks to both the people I worked with at SGI and and DEC's CRL. I would especially like to thank Luis Stevens and Jeff Heller who were willing to spend time working through many issues, who showed patience with my early lack of concern for implementation details, and who, in the process, became good friends.

Part of what has made Rochester a great experience has been the many friends and sparring partners I have had during my tenure. I owe thanks to many: to

Leonidas, who has been a great friend and excellent co-researcher, willing to listen to and help evaluate ideas both wild and sane; to Kirk, for our special friendship, for reminding me how fun and exciting being young and learning can be, and for helping me remember there's more to life than experiments, papers, and numbers; to Ricardo, who provided many interesting conversations and was invaluable to discuss possible ideas with; to Tom and his family for their friendship; to my tennis partners: Olac, Tom, and Jeff; my chess partners: Tom, Ray, and Srini; my go partners: Patrice (mentor), Jeff, Andrew, Feng, Wu, and Dana; and the numerous people who challenged me in hockey, swimming, and other activities that kept me in shape, mentally and physically, and feeling alert when I returned to research; to all these and more, thank you. I would also like to thank my brother, who throughout my life has supported my efforts.

At Rochester we are fortunate to have excellent systems support in maintaining our environment and allowing us to concentrate on research. Special thanks to Liudy for the countless hours he spent working with me on the Maxware hardware for the shepherding application and anything else I ever needed along the way. I also owe thanks to the astute readers of our earlier papers: to Hiroaki Takada and to Injong Rhee and Chi-Yung Lee for discovering timing windows in the Smart Queue algorithm. Thanks also to Donna Bergmark and the Cornell Theory Center for their help with the KSR.

This research work was supported in part by: an ARPA Fellowship in High Performance Computing administered by the Institute for Advanced Computer Studies, University of Maryland, NSF Research Grants CDA-8822724 (Institutional Infrastructure grant), IRI-8903582, CCR-9005633, and CCR-9319445, by DARPA Research Grant MDA972-92-J-1012, and by ONR contracts number N00014-93-1-0221 and number N000014-92-J-1801 (in conjunction with the ARPA Research in Information Science and Technology–High Performance Computing, Software Science and Technical program, ARPA Order no. 8930). Experimental results were obtained in part through use of resources at the Cornell Theory Center, which receives major funding from NSF and New York State; additional funding comes from ARPA, the NIH, IBM Corporation, and other members of the Center's Corporate Research Institute.

Abstract

Designing high performance parallel applications is a challenging task. The task becomes more difficult when these applications are on a multiprogrammed multiprocessor or function in the real world and need to meet real-time constraints. From the application's perspective these environments are unpredictable: under multiprogramming it is possible to be switched out at any time, and in the real world unexpected events can occur. Our thesis is that by sharing information across the traditional system-application interface we can achieve high performance in parallel applications both in the presence of multiprogramming and when meeting real-time constraints. The implemented techniques we describe not only perform better, but are simpler than those that would be required without such interaction. The contributions of this thesis, which are stated explicitly later, lie both in the mechanisms and techniques themselves and in the paradigm and implementation of the runtime environment that allows the mechanisms to exist.

The mechanisms and techniques described throughout this thesis respond and adapt to unpredictable environments, whether caused by an unanticipated real-world event or an unexpected context switch. Synchronization is needed in almost all parallel programs, and, if oblivious to scheduler decisions, can have deleterious effects on application performance. We describe a set of *scheduler-conscious* and *preemption-safe* synchronization techniques that address the possibility of preemption on multiprogrammed multiprocessor machines. Mechanisms used in responding to the real world need to be flexible and adaptive. We describe a set of such mechanisms that allow real-time programs to adapt effectively to a changing and unpredictable world. We also describe a model for structuring the entire system (application, runtime, and kernel) in real-world environments.

Table of Contents

Curriculum Vitae ii

Acknowledgments iii

Abstract v

List of Figures viii

1 Introduction xii

1.1 Background 2

1.2 Problems 3

1.3 Programs Interacting With the Real World 4

1.4 Programs Interacting with Other Programs: Synchronization and Multiprogramming 6

1.5 Contributions 9

1.6 Roadmap 10

2 Ephor: A Runtime for SPARTAs 12

2.1 Related Work 15

2.2 A Model of SPARTAs 16

2.3 Planner Design 18

2.4 Supporting Results 23

2.5 Runtime Support 27

2.6 Layer Responsibilities 29

2.7 Implementation 32

2.8 Summary 34

3 Synchronization in the Presence of Multiprogramming **35**
3.1 Introduction 36
3.2 Background 39
3.3 Solution Structure 44
3.4 Algorithms 47
3.5 Results 54
3.6 Synchronization Conclusions 70

4 Scheduling **72**
4.1 Policies 74
4.2 Analysis 76
4.3 Implementation and Mechanisms 83
4.4 Performance and Results 87
4.5 Scheduling Conclusions 97

5 Ephor Mechanisms **98**
5.1 A Set of Mechanisms 99
5.2 Dynamic Technique Selection 105
5.3 Dynamic Parallel Process Control 109
5.4 Mechanisms Conclusions 111

6 Shepherding: An Example Application **112**
6.1 Introduction 113
6.2 Specifics of the Hardware and Software 116
6.3 Decomposition and Real-Time Control 120
6.4 Vision 125
6.5 Manipulation 133
6.6 Example Application Conclusions 134

7 Conclusions **143**
7.1 Contributions 143
7.2 Future Directions 145

Bibliography **147**

List of Figures

1.1 Without a runtime environment (like Ephor) there exists a conceptual (kernel does not know high-level constructs such as goals) and a mechanism (kernel cannot reasonably allocate needed resources and has to return a failure) gap between the application-level abstractions and kernel-level mechanisms is difficult to bridge. . . . 6
1.2 Queued mutex lock performance 7

2.1 The real-world shepherding application (camera overhead) 15
2.2 The three layers in the design of a real-world application 17
2.3 Program structure 18
2.4 Application-Ephor interface 24
2.5 Adapting to high fixed loads 25
2.6 Adapting to high variable loads 25
2.7 Fixed versus adaptable parallel planners 26
2.8 The three layers in the design of a real-world application 30
2.9 Layer information exchange 30

3.1 Pseudocode declarations for the kernel-application interface. . . . 47
3.2 Varying multiprogramming level on an 11-processor SGI Challenge. 58
3.3 Varying multiprogramming level on a 63-processor KSR 1. 58
3.4 Varying the number of processors on the SGI Challenge with a multiprogramming level of 2. 58
3.5 Varying the number of processors on the KSR 1 with a multiprogramming level of 2. 58
3.6 Completion time (secs) for Cholesky on the SGI using 11 processors (multiprogramming level = 2). 60
3.7 Completion time (secs) for Cholesky on the KSR using 63 processors (multiprogramming level = 2). 60

3.8 Completion time (secs) for Quicksort on the SGI using 11 processors (multiprogramming level = 2). 60

3.9 Completion time (secs) for Quicksort on the KSR using 63 processors (multiprogramming level = 2). 60

3.10 Latency (acquire + release) of mutual exclusion locks on the SGI. The extra numbers in the right-hand column indicate the absolute and percentage increase in latency of the preemption-safe and scheduler-conscious locks with respect to their scheduler-oblivious counterparts. 62

3.11 Latency (acquire + release) of reader-writer locks on the SGI. The extra numbers in the second and third columns indicate the absolute and percentage increase in latency of the preemption-safe and scheduler-conscious locks with respect to their scheduler-oblivious counterparts, for both the reader and writer parts of the algorithms. 62

3.12 Varying the multiprogramming level for the reader-writer lock on the SGI (11 processors). 63

3.13 Varying the multiprogramming level for the reader-writer lock on the KSR (63 processors). 63

3.14 Varying the number of processors for the reader-writer lock on the SGI (multiprogramming level = 2). 63

3.15 Varying the number of processors for the reader-writer lock on the KSR (multiprogramming level=2). 63

3.16 Performance of the small-scale barriers for the synthetic program. 66

3.17 Gaussian Elimination run-time for different barrier implementations (multiprogramming level=1). 67

3.18 Gaussian Elimination run-time for different barrier implementations (multiprogramming level=2). 67

3.19 SOR run-time for different barrier implementations (multiprogramming level=1). 68

3.20 SOR run-time for different barrier implementations (multiprogramming level=2). 68

3.21 Varying the number of processors for the barriers on the KSR (multiprogramming level = 2). 68

3.22 Varying the frequency of repartitioning decisions for the barriers on the KSR (57 processors). 68

3.23 Gaussian elimination run-time on the KSR using 57 processors (multiprogramming level = 1) 69

3.24 Gaussian elimination run-time on the KSR using 57 processors (multiprogramming level = 2) 69

4.1 Comparison of the scheduling mechanisms 74
4.2 Execution times for the vision processing task 78
4.3 Gaussian distribution with I set assuming no offset and no slack . 79
4.4 Gaussian distribution with an offset but no slack 79
4.5 Slack times for many tasks 80
4.6 Slack times for few tasks 80
4.7 Gaussian distribution for execution and slack times 80
4.8 Summed Gaussian distribution with offset slack 80
4.9 Time distribution examples 82
4.10 Example processor 7 utilization 89
4.11 Example processor 4 utilization 89
4.12 Example average processor utilization 90
4.13 Average processor utilization with few tasks and low load 91
4.14 Average processor utilization with many tasks and low load 91
4.15 Average processor utilization with moderate tasks and low load . 91
4.16 Processor 5 utilization with moderate tasks and low load 91
4.17 Average processor utilization with few tasks and high load 92
4.18 Average processor utilization with moderate tasks and low load . 92
4.19 92
4.20 Average processor utilization with many tasks and high load increased period 92
4.21 Average processor utilization from real program with few tasks . . 93
4.22 Average processor utilization from real program with many tasks . 94
4.23 Processor 4 utilization for real program with many tasks 95
4.24 Processor 5 utilization for real program with many tasks 95
4.25 Processor 6 utilization for real program with many tasks 95
4.26 Processor 7 utilization for real program with many tasks 95
4.27 Number of sheep confined with few tasks in system 96
4.28 Number of sheep confined with many tasks in system 96

5.1 Adapting to high fixed loads. 107
5.2 Adapting to high load, averaged variable loads. 107
5.3 Adapting to high fixed priorities. 108
5.4 Adapting to high priorities, averaged variable priorities. 108

5.5 Fine adjustment between similar fixed loads. 108
5.6 Fine adjustment between similar averaged loads. 108
5.7 Fixed versus adaptable parallel planners: low demand. 110
5.8 Fixed versus adaptable parallel planners: medium demand. 110
5.9 Fixed versus adaptable parallel planners: high demand. 111

6.1 Side view of a Lego sheep. 114
6.2 Top view of a Lego sheep. 114
6.3 A flow diagram of the shepherding project. 114
6.4 Code added to kernel to allow it to map in VME space 118
6.5 Code establishing pointers to VME space for direct access of boards 118
6.6 C code for the ephor_begin_proc macro 136
6.7 C code for the ephor_end_proc macro 137
6.8 A covered Lego sheep. 137
6.9 A sheep and cover. 137
6.10 The actual image (on the right) and its shadow array (on the left) 137
6.11 The lines created by the quick circle method 138
6.12 Comparing blobify to trackify 138
6.13 A sample scene with two sheep and their velocity vectors. 138
6.14 Velocity for a stationary sheep 139
6.15 The actual measured velocity of the toy sheep where the x-axis is in twentieths of a second and the y-axis is pixels per second. . . . 139
6.16 The parameters used in both of the $\alpha - \beta$ filters. 140
6.17 The output of the first $\alpha - \beta$ filter. 140
6.18 The output of the second $\alpha - \beta$ filter. 140
6.19 The measured, first filter, and second filter velocities. 141
6.20 Error in predicting a sheep position. 141
6.21 Shepherding setup 142

1 Introduction

Designing high-performance parallel applications is a challenging task. The task becomes more difficult when these applications are run on a multiprogrammed multiprocessor or when they function in the real world and need to meet real-time constraints. Researchers have used advances in hardware technology to design larger and more complex real-time applications. Larger applications require new integration techniques while more complex applications require a restructuring of the underlying system support. We examine the system design issues of supporting SPARTAs (Soft PArallel Real-Time Applications). From the application's perspective these parallel or multiprogrammed environments are unpredictable: under multiprogramming it is possible to be switched out at any time, and in the real world unexpected events can occur. Our thesis is that *by sharing information across the traditional system-application interface we can achieve high performance in parallel applications both in the presence of multiprogramming and when meeting real-time constraints.* The implemented techniques we describe not only perform better, but are simpler than those that would be required without such interaction. The contributions of this thesis, which are stated explicitly later, lie both in the mechanisms and techniques themselves and in the paradigm and implementation of the runtime environment that allows the mechanisms to exist.

The mechanisms and techniques described throughout this thesis respond and adapt to unpredictable environments, whether caused by an unanticipated real-world event or an unexpected context switch. Synchronization is needed in almost all parallel programs, and, if oblivious to scheduler decisions, can have deleterious effects on application performance. We describe a set of *scheduler-conscious* and *preemption-safe* synchronization techniques that address the possibility of preemption on multiprogrammed multiprocessor machines. Mechanisms used in responding to the real world need to be flexible and adaptive. We describe a set of such mechanisms that allows real-time programs to adapt effectively to a changing and unpredictable world. We also describe a model for structuring the entire system (application, runtime, and kernel) in real-world environments.

1.1 Background

When a program is confronted with unexpected events it is important for it to be able to adapt. If the program cannot adapt, sometimes an error in program correctness will result. More often though, the program's performance will suffer. This thesis will address two primary sources of unpredictability: programs interacting with each other and programs interacting with the real world.

1.1.1 Programs Interacting With the Real World

In the real world unexpected events can occur. To respond to these events a program needs to be able to adapt. Even in periods when unexpected events do not occur there will still be times of lower demand and times of higher demand. When responding to unexpected events, it may be necessary to adapt to achieve program correctness, whereas the inability to adapt to periods of differing demand will most likely impact performance. SPARTAs that unable to adapt to varying resource demand will not perform as well as those that can.

A lot of previous work in real-time systems focused on hard real-time issues inherently producing approaches that are not adaptive. The need for adaptive techniques becomes clear when analyzing the requirements and demands of soft real-time systems. Some research groups did focus on soft real-time application and designed adaptive applications [Hal90]. However, their techniques often only applied to the particular application they examined; the intent of our work is to design general adaptive mechanisms. To address the issue of the static and immutable schedules, researchers proposed the idea of slack stealing [TL94] and anytime algorithms [LLS$^+$91; WHR90]. The difficulties with previous approaches is that they either do not generalize adequately or do not go far enough in providing adaptive mechanisms for the emerging class of soft real-time applications that are the focus of our work.

1.1.2 Programs Interacting With Other Programs

When applications share a machine it becomes necessary to multiplex the processors amongst the different applications. This will result in an application's process being context-switched out. If the application does not take into account scheduler actions performance degradation will occur when processes attempt to synchronize with their sleeping peers. This performance loss is especially pronounced on scalable multiprocessors. For locks the performance loss is due to other processes spinning while a preempted peer is in the critical section, and for barriers the performance loss occurs when a process spins at a barrier while other peer processes are preempted and could be making better use of its processor.

Some research groups have addressed the issue of preemption in the critical section. They either avoid it [ELS88; MSLM91] or recover once it occurs [ABLL92; Bla90]. These techniques go a long ways towards addressing the difficulties encountered by locks on small machines, but are not sufficient for scalable locks. For barrier based programs untimely preemption is not an issue. Rather, what is needed is the ability to yield a process to a peer and more importantly to know when to yield a processor. While Black [Bla90] provides a mechanism for suggesting to the kernel to run a peer process, this is not enough as a process also needs to know *if* it should perform the yield. We address these barrier issues for both bus-based multiprocessors and for scalable multiprocessor where a reconfigurable tree barrier is required.

1.2 Problems

In obtaining high-performance parallel applications that interact with the real world or that use synchronization on multiprogrammed multiprocessors a number of difficulties arise. The thesis proposes solutions to the following problems:

Programs Interacting with the Real World:

1. There exists a gap between the existing real-time kernel mechanisms and the functionality desired by a SPARTA programmer.
   (a) Hard real-time systems have not been designed with adaptability needed to support the newly evolving soft real-time applications.
   (b) The tools and systems support for parallel applications have been structured around scientific applications and are not well suited for SPARTA environments.
2. It is not clear what the correct interface to robotics, real-time artificial intelligence, vision, etc. should be, and what functionality should be in the operating system level and what in the application level.
3. Real-time scheduling mechanisms are not well suited for SPARTAs.

Programs Interacting with Other Programs:

4. Multiprogramming can have a deleterious effect on parallel applications written in a *scheduler oblivious* manner, even programs well-designed for dedicated machines. Under a scheduler-oblivious multiprogramming model, an application's processes are context switched out unexpectedly and without the application's other peer processes aware that it happened.

5. Scalable synchronization algorithms based on distributed data structures are particularly susceptible to multiprogramming effects.

1.3 Programs Interacting With the Real World

There exists a gap between the existing real-time kernel mechanisms and the functionality desired by a SPARTA (Soft PArallel Real-Time Application) programmer. In order for programs to function in the real world they have to meet real-time constraints and be prepared to handle unexpected events. Depending on the specifications for an application, it may be classified as a soft or a hard real-time application. Loosely, hard real-time applications are characterized by requiring absolute guarantees and often exist in environments where the price of failure is high (e.g., airplane autopilot), while soft real-time applications have the flexibility to occasionally take longer than projected (e.g., mail delivery robot). In the last decade there has been a shift in the type and complexity of applications designed for the real-world: systems are needed that allow soft and hard real-time components to coexist.

Many real-world applications contain both hard and soft real-time components. There has been considerable work on hard real-time system design such as [SR87] as well as work for parallel [HH89] and distributed [SGB87] environments. Target applications for hard real-time systems include airplane autopilot or nuclear power plant control. New complex, parallel soft real-time applications have been generating considerable interest [Hal90; DJ86; Sha87; DHK+88; KP93]. Some example applications are: autonomous navigation, reconnaissance, and surveillance; operator-in-the-loop simulation; and teams of autonomous cooperating vehicles.

Designing a SPARTA is challenging, since in its full generality it calls for dynamic decision making about resource allocation, scheduling, choice of methods, handling reflexive or reactive behavior smoothly within a context of planned or intended actions, and a host of other issues not typically encountered either in offline or hard real-time applications. SPARTAs need different system support than either the large data-crunching scientific programs or the smaller less-structured applications currently being investigated in parallel environments. Further, supporting such applications was beyond the intended scope of previous real-time kernels because other more fundamental or lower level issues needed to be addressed first.

It is not clear what the correct interface to robotics, real-time artificial intelligence, vision, etc. should be, and what functionality should be in the operating system level and what in the application level. The real-time community has acknowledged the need to explore issues raised by SPARTAs. Stankovic [Sta92] enumerates a number of key issues that we have addressed in the design of Ephor[1] our runtime environment designed to support SPARTAs. Among them are "what are the correct interfaces to robotics, RTAI, Vision, ...etc." and "what functionality should be in the OS level and what in the application level."

The problem is illustrated in Figure 1.1. There is a large conceptual and mechanism gap between a typical SPARTA and a typical real-time operating system. As indicated in Figure 1.1, the conceptual gap occurs when the kernel doesn't understand high-level constructs, such as a goal, and a mechanism gap occurs when the kernel can not reasonably allocate needed resources and has to return a failure. Therefore, an integral part of supporting SPARTA design will be providing an intermediate runtime layer. In addition to proposing an appropriate model for SPARTA design, we describe our experiences building Ephor, including what motivated its conception and development, and the resulting separation of responsibilities both easing the design of SPARTAs and improving their performance.

Our runtime, Ephor, interacts with SPARTAs, maintaining hard real-time behavior when needed while providing graceful degradation in cases in which performance is important but not critical to the success of the application. Our intermediate runtime layer is built on a hard real-time substrate providing the additional functionality needed by SPARTAs. The runtime reduces the replicated work of system monitoring and dynamic decision-making that is common between applications.

Initially, the effort needed to develop a general run-time package, such as Ephor, versus simply incorporating the needed portions into an application, may appear prohibitive. However, an analogy to threads of control indicates this may not be so. Historically, threads of control were thought of simply as support for co-routining under direct user control. However, through time, many other issues with thread management have arisen. A similar situation applies with tasks, methods, or even planners in SPARTAs. Synchronization between tasks may be a significant issue, as might be the interleaving of tasks, or running one based on an exception generated by another, etc. Although it is conceivable these issues could be handled at the application level much the same way parallel thread management could be, there are compelling reasons for studying the systems aspects of such general capabilities.

Real-time scheduling mechanisms are not well suited for SPARTAs. Hard real-time systems have not been designed to support the newly evolving soft real-time applications. In particular, they lack the flexibility needed to adjust to a complex and dynamic environment. The reason is that they must provide absolute predictability and guaranteed scheduling. As part of designing Ephor we have examined scheduling, a core area of real-time research. We have invented and

[1]Ephor was the name of the council of five in ancient Greece that effectively ran Sparta

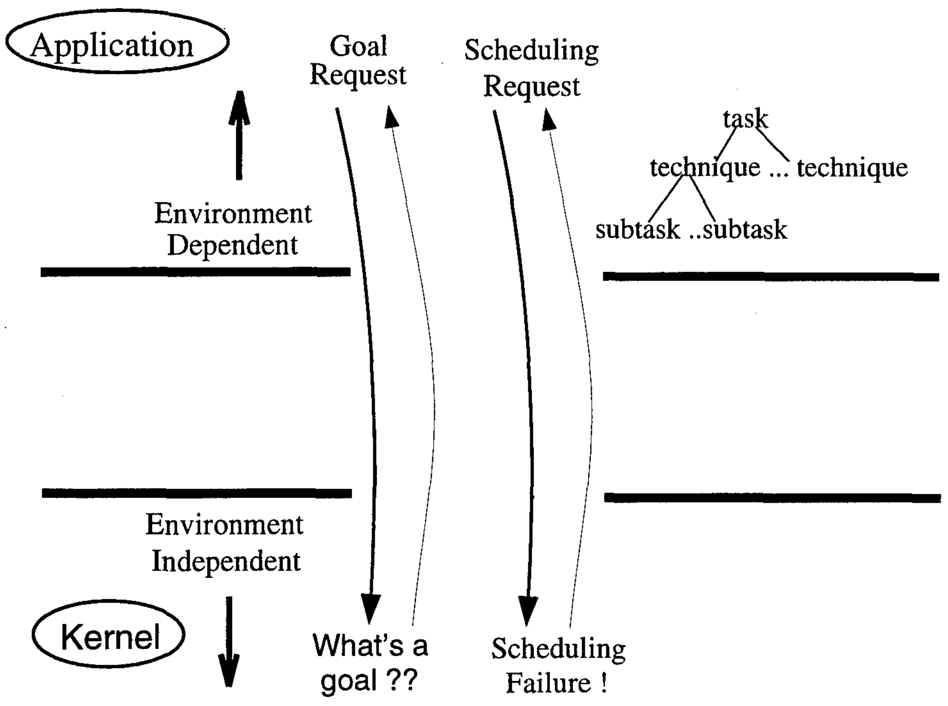

Figure 1.1: Without a runtime environment (like Ephor) there exists a conceptual (kernel does not know high-level constructs such as goals) and a mechanism (kernel cannot reasonably allocate needed resources and has to return a failure) gap between the application-level abstractions and kernel-level mechanisms is difficult to bridge.

analyzed new scheduling paradigms and incorporated them into Ephor. These scheduling mechanisms are particularly suited for SPARTA environments. Here again we have achieved better performance by matching the specifications of the scheduling mechanisms to the model of tasks requesting them. Two recurrent themes throughout our work for real-world applications are the importance of the ability for the system (runtime and kernel) to adapt to changing application needs, and the benefit of widening the interface between the typical kernel and application in a SPARTA environment to facilitate this desired flexibility.

1.4 Programs Interacting with Other Programs: Synchronization and Multiprogramming

Multiprogramming can have a deleterious effect on parallel applications written in a scheduler-oblivious manner, even programs well-designed for dedicated machines. Multiprogramming means that an application shares the machine with one or more other applications. There are different multiprogramming models, but assuming a dynamic model (applications can come and go) the view of the machine to a given application will change over time. *Under a scheduler-oblivious model, an application's processes are context switched out unexpectedly and without the application's other peer processes being aware that it happened.* Another multiprogramming model partitions the machine into smaller "machines" and doles out those smaller "machines" to requesting applications. In the first model processes are context switched to allow another application's process to run while in the second model processes are context switched to allow another process within the same application to run. In either case performance can suffer.

In our experiments, we have found that algorithms that provide excellent performance in the absence of multiprogramming may perform orders of magnitude worse when multiprogramming is introduced. These results suggest the need for *scheduler-conscious* synchronization techniques. Figure 1.2 shows the performance of two programs, one modified to use our scheduler-conscious synchronization techniques and the other not. Although the latter algorithm outperforms any previous synchronization algorithm on a dedicated machine, the figure shows that multiprogramming can have a disastrous effect on applications even well designed for dedicated machines. Clearly, providing scheduler-conscious preemption-safe mechanisms is important.

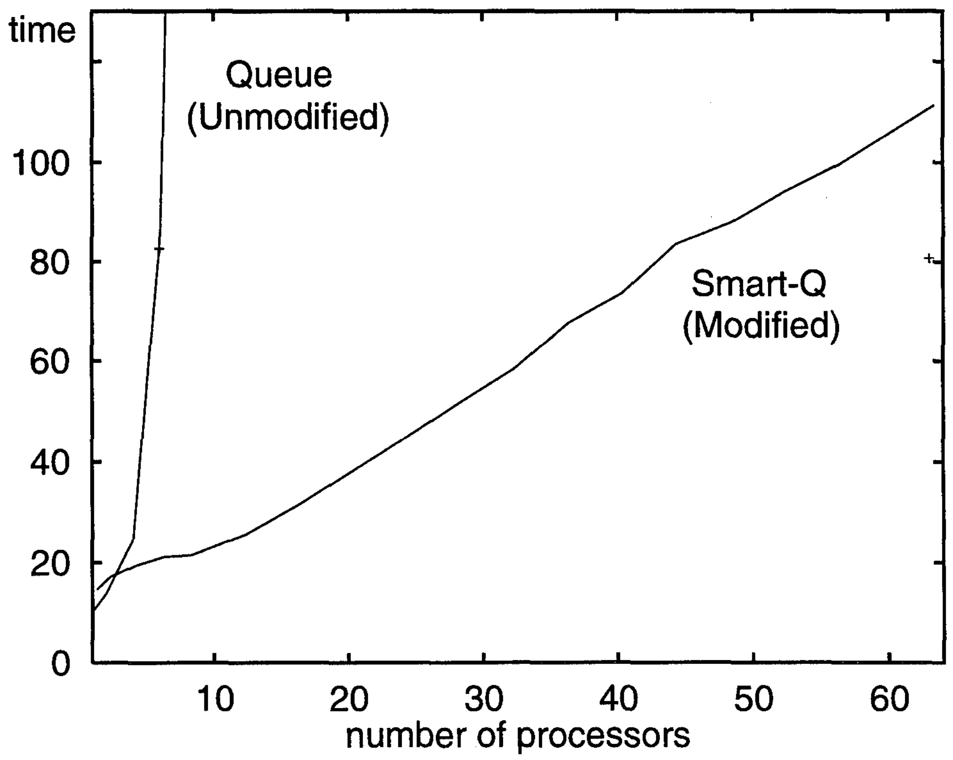

Figure 1.2: Queued mutex lock performance

A fundamental question for a synchronization mechanism is whether a process upon reaching a synchronization point should block (yield the processor to another) or spin (repeatedly test the desired condition). Spinning makes sense when the expected wait time of a synchronization operation is less than twice the context switch time, or when the spinning processor has nothing else useful to do. Otherwise it is better to block. Spinning in user-level code tends to work well only if each process runs on a separate physical processor. If the total number of processes in the system exceeds the number of processors, then some processors will have to be multiprogrammed. The processes on a given processor may be from different applications or, if the scheduler partitions the machine, from a single application. In either case, conflicts between scheduling and synchronization can seriously degrade performance when:

- a process is preempted while holding a lock,
- a process is preempted while waiting for a lock and then is handed the lock while still preempted, or
- a process spins when some preempted process could be making better use of the processor.

Scalable synchronization algorithms based on distributed data structures are particularly susceptible to multiprogramming effects. Most synchronization algorithms have been designed to run on a dedicated machine, with one application process per processor, and can suffer serious performance degradation in the presence of multiprogramming. Problems arise when running processes block or, worse, busy-wait for action of a peer process that the scheduler has chosen not to run. We show that these problems are particularly severe for scalable synchronization algorithms based on distributed data structures. This is because those data structures often impose a predetermined ordering on the expected progression through the synchronization mechanism, and the unexpected actions of the scheduler causes significant additional problems because of this fixed ordering.

We describe and evaluate a set of algorithms that perform well in the presence of multiprogramming while maintaining good performance on dedicated machines. We consider both large and small machines, with a particular focus on scalability, and examine mutual-exclusion locks, reader-writer locks, and barriers. The algorithms we study fall into two classes: preemption-safe techniques that decide whether to spin or block on the basis of heuristics, and scheduler-conscious techniques that use information from the scheduler to drive more accurate decisions.

1.5 Contributions

In this thesis we show that by sharing information across the kernel-application boundary, effectively widening that interface, we achieve better performing techniques and mechanisms with simpler implementation. Our contributions lie both in the techniques and mechanisms as well as in the model for the interface.

There have been research groups in the past that have considered real-world applications [Hal90] embodying some of the design qualities of Ephor but in an application specific way. Other researchers [BS91] have looked at providing adaptive real-time systems. There has also been a tremendous amount of work on scheduling for real-time systems, extending from early rate monotonic work [LL73] to more recent multiprocessing work [BDW86]. Ephor incorporates some of the past successful techniques and our new high performance techniques into a single runtime package that works across applications. In addition to the useful new techniques, we describe how to combine them in a general runtime environment for SPARTAs.

Another aspect of information sharing is scheduler-conscious synchronization. Research groups have shown how to avoid preempting a process that holds a `test_and_set` lock [ELS88; MSLM91], or to recover from this preemption if it occurs [ABLL92; Bla90]. Other groups have developed heuristics that allow a process to guess whether it would be better to relinquish the processor, rather than spin, while waiting for a lock [Ous82; KLMO91]. Our synchronization interface builds on ideas from previous work with a few additional extensions. However, our primary contributions are the techniques we develop that take advantage of this interface to provide scheduler-conscious synchronization. Throughout the thesis we show that by sharing information across the kernel-user interface we can ease the design of synchronization algorithms and SPARTA mechanisms and improve their performance.

To solve the problems presented in Section 1.2 this thesis provides the following set of contributions:

Programs interacting with the real world:

1. Mechanisms to fill the gap between existing mechanisms and those desired by a SPARTA programmer.
   (a) A dynamic technique selection mechanism [WB93].
   (b) A suite of mechanisms suitable for SPARTAs: dynamic parallel process control, de-scheduling of all tasks linked to a goal, access functions for sharing information, optional partitioning of soft and hard real-time tasks, overdemand detection and recovery, and early termination of tasks.

2. A methodology for designing SPARTAs.

    (a) An application independent methodology for incorporating mechanisms into a SPARTA runtime environment [WB94].

    (b) A set of recommendations for task design to SPARTA programmers [WB95].

3. A set of scheduling policies designed for SPARTA environments [WB96].

Programs interacting with other programs:

4. Scheduler-conscious synchronization techniques: a scheduler-conscious barrier for small machines (in which a centralized data structure does not suffer from undue contention, and in which processes can migrate between processors) [KW93; KWS94].

5. Scheduler-conscious synchronization techniques targeted for scalable machines.

    (a) A preemption-safe ticket lock and scheduler-conscious queue lock, both of which provide FIFO servicing of requests and scale well to large machines [WKS94; KWS94].

    (b) A fair, scalable, scheduler-conscious reader-writer lock (the non-scalable version is trivial) [KWS94].

    (c) A scheduler-conscious barrier for large machines that are partitioned among applications, and on which processes migrate only when repartitioning occurs [WKS95; KWS94].

1.6 Roadmap

Chapter 2 starts by describing the difficulties in designing SPARTAs. We present our shepherding application, which embodies characteristics of the real-world applications studied in this thesis. From this discussion we show the importance of designing adaptable applications. We then describe a set of recommendations for SPARTA programmers to achieve better performing applications (Problem and Contribution 2b from Sections 1.2 and 1.5). This sets the stage for a description of the underlying support needed to implement these recommendations. With this we segue into the design of Ephor, our application independent runtime environment for SPARTAs (Problem and Contribution 2a).

Since this thesis covers considerable ground, related work is discussed in each chapter where appropriate. Also, results pertinent to the topics discussed are presented on a chapter by chapter basis. Following this philosophy, Chapter 3 presents previous efforts to handle multiprogramming in multiprocessors and then presents our scheduler-conscious synchronization algorithms (Problem and Contributions 4 and 5). Continuing in Chapter 4 we present a sampling of the real-time scheduling work and then describe, analyze, and present results from our suite of scheduling policies targeted at SPARTA environments (Problem and Contribution 3).

In Chapter 5 we describe the advantages of dynamic technique selection (Problem and Contribution 1a) and give results, both qualitative and quantitative, indicating its benefits. In this chapter we also present the other mechanisms we developed in Ephor (Problem and Contribution 1b). The results from the scheduling policies and dynamic technique selection and other techniques come primarily from a shepherding simulator we have developed to provide us a representative sample SPARTA. In Chapter 6 we describe our experiences putting everything together and we describe the design of a real world shepherding application in our robotics laboratory. We restate our contributions and provide concluding remarks in Chapter 7. We leave the reader with a host of possible future directions to pursue based on the work we presented in the thesis.

2 Ephor: A Runtime for SPARTAs

Designing a SPARTA (Soft PArallel Real-Time Application) requires the combination of three different components that are difficult to construct even in isolation. SPARTAs are parallel programs, real-world planners, and real-time programs combined into one application. Each of these elements is a challenging problem in its own right. With the added difficulty of effectively combining them, achieving a well performing and correct SPARTA could be overwhelming without the help of tools and guidelines to automate and constrain the process. In Chapters 4 and 5 we present in detail the specific mechanisms designed and implemented in Ephor. In this chapter we describe a set of recommendations for implementing SPARTAs and a methodology for designing underlying runtime and system support. The goal is to provide the programmer with tools and methods that allow a SPARTA to utilize a multiprocessor most effectively.

If a SPARTA programmer is oblivious to real-time issues when designing an application, poor or incorrect behavior may result. Tension arises due to the discrepancy between how a programmer wants to design an application (typically not at the level of real-time considerations) and the requirements of planning and acting in the real world. We have developed Ephor to support SPARTA development and execution. Our goal is to remove resource (including time) management decisions from the user so that the standard techniques available for designing applications such as intelligent robotic applications can be applied to SPARTAs.

In the first half of this chapter we focus on implementing effective planners[1] in parallel real-world applications. Previously, designing a planner for a SPARTA meant tracking resource allocation, timing tasks, and handling other concerns of interacting in the real world. The combination of Ephor and our model of planning in SPARTAs considerably simplifies design.

[1] "Planning" refers to all forms of cognitive reasoning, problem-solving, and decision-making techniques for deciding what to do next, from simple random choice through sophisticated modern planners.

A key principle that underlies our work that we leverage throughout our discussion is that in a dynamic real-world environment it is important to be able to adapt. While this may be intuitive, its implications for planner (and runtime) design are significant. The importance of adapting holds both for the action taken by the application as well as how the application decides on that action. More concretely, in later sections we discuss the advantages of having several planners (with the same *task*) varying in resources consumed (and thus quality of result). This diversity is useful because it allows an adaptive decision to be made during execution when the application needs a particular *task* solved. In part, the principles for designing SPARTA planners are motivated by what tools and mechanisms the underlying runtime environment and operating system can provide to the application programmer. Creating a happy marriage between what can be supported from the system's point of view, and what model the real-world programmer would like, is important to successfully implement real-world applications.

Throughout, we use the specific application domain of shepherding to provide concrete examples of our principles, and to demonstrate their effectiveness. The shepherding application domain is flexible and maps onto a large class of real-world applications that involve uncertain actions, uncertain sensing, real-time constraints and responsibilities, planning and replanning, dynamic resource management, dynamic focus of attention, low-level reflexive behaviors, and parallel underlying hardware (e.g. purposive vision, autonomous vehicle control and navigation). A real-world shepherding implementation, described in Chapter 6, runs in our robotics laboratory [BB92][vKW94](see Figure 2.1), but the results in Section 2.4 are from a real-time simulator that allows greater flexibility in experimentation. The implementation consists of self-propelled Lego vehicles ("sheep") that move around the table ("field") in straight lines but random directions. An overhead camera allows visual monitoring of the sheep's progress. Each sheep moves at constant velocity until herded by the robot arm ("shepherd"), which redirects it towards the center of the field. A second robot arm ("wolf") can encroach on the field and remove ("kill") sheep if not prevented. The shepherd has a finite speed and can affect only one sheep at a time. The goal of the shepherd is to keep as many sheep on the table as possible, and the more powerful the sheep behavior-models and look-ahead, the better the results.

General approaches to designing SPARTAs are only now beginning to emerge, and usually individual solutions do not generalize well. We believe this is a two part problem. First, underlying runtime and system support (Ephor) is needed. Secondly, the principles in designing effective planners for these types of applications must be understood. To better motivate the need for underlying runtime support mechanisms discussed in chapters 4 and 5 we present a set of principles for SPARTA planner design that yield a better structured application, simplify design, and improve performance. We start by providing a model of SPARTAs in Section 2.2. Section 2.3 lists and describes in detail the principles involved in

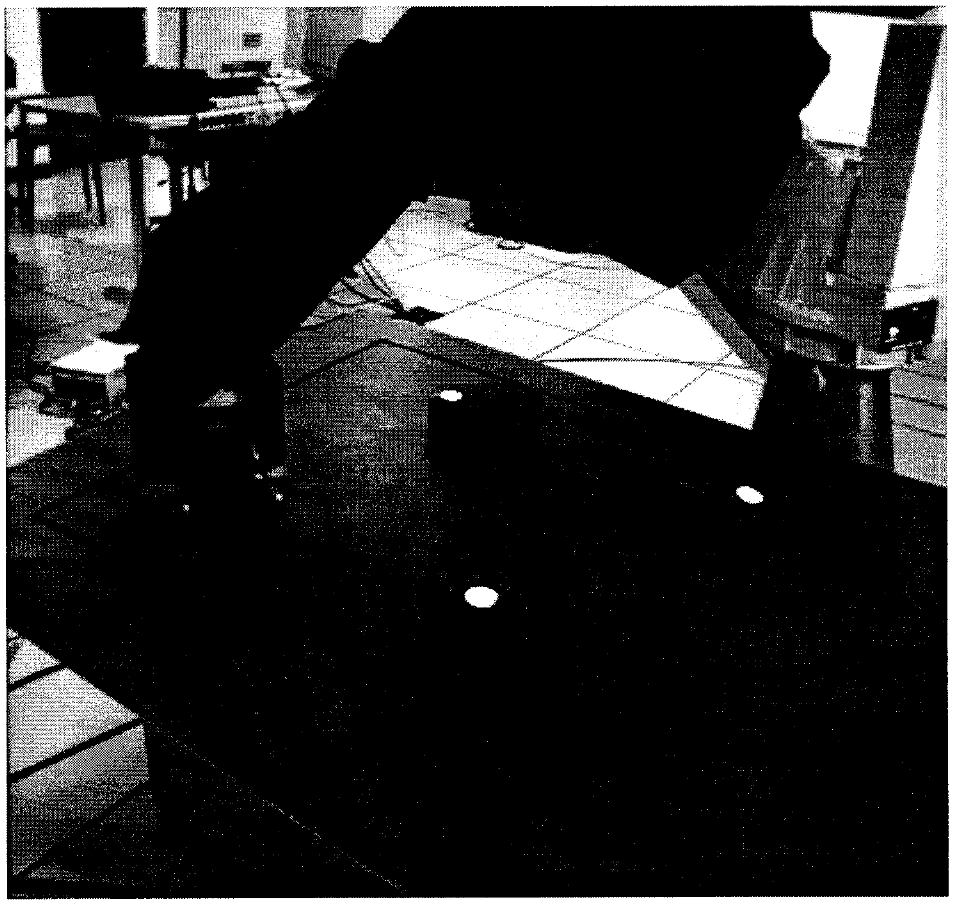

Figure 2.1: The real-world shepherding application (camera overhead)

designing effective planners for SPARTAs. We then provide results demonstrating the effectiveness of the planner principles. After describing recommendations for the application layer we present our model for complete SPARTA design. In Section 2.5 we briefly review motivating factors and SPARTA properties. Section 2.5.3 discusses the design problems for a SPARTA. Based on those difficulties we demonstrate the need for a division of responsibilities split among three layers in a SPARTA. In Section 2.6 we outline the responsibilities of each layer and provide arguments as to the benefits gained by such a layering. The last section (2.7) of this chapter describes the important aspects of our implementation of Ephor.

2.1 Related Work

Stankovic [Sta92] has pointed out some shortcomings of existing real-time systems in supporting complex applications. Ephor addresses some of these concerns.

Nirkhe and Pugh [NP91] have made clear that current real-time techniques provide a very limited environment, e.g. disallowing recursion, while loops, dynamic memory allocation, concurrency, and synchronization. Their work attempts to allow the programmer to use these constructs by a method called partial evaluation.

Chodrow, *et al* [CJ91] have presented a method to monitor real-time systems and suggested that this information could be used to adapt in a dynamic environment. Haban and Shin [HS89] have used information obtained by monitoring real-time systems to schedule tasks with random execution times. Other work on monitoring real-time systems was performed by Tsai [TFC90]. Although this work was for distributed systems, we foresee similar requirements in a parallel environment.

Oliveira *et al* [OQC91] have shown how dynamically to choose and decompose a task under a blackboard style architecture. They then, however, may reject a task based on other requirements in the system. Smith and Setliff [SS91] have shown how to automatically synthesize a real-time program from a description of task-level timing constraints and functional and behavioral descriptions of the processing for each task. However, the tasks are coarse-grained, and designing a general-purpose program is impossible. Davis, *et al* [DPAB95] have also proposed an architecture similar to the three teered structure we propose in the this section. They also propose flexible scheduling mechanisms for the real-time domain.

Marsh *et al* [MSLM91] [MBL$^+$92] developed the idea of first class user-level threads, which are an example of improved system-application communication. That work is at the operating system level, but some of its constructs are clearly useful to applications (notification of impending preemption, for instance).

There has been an abundance of work both on scheduling in real-time systems and on designing planners. Scheduling work such as [ZRS87] or [RSS90] would be useful in our underlying hard real-time substrate, but is both too restrictive and pessimistic (assumes worst-case behavior) for the run-time's scheduling of techniques. Adaptive planning work such as [HHP92] or [GD92] is effective in handling environmental factors. Our run-time allows sophisticated schedulers to take into account internal system state.

2.2 A Model of SPARTAs

This section is devoted to discussing models for designing SPARTAs. We describe our model and its applicable domains. There are two disparate approaches to designing real-world AI and robotic applications. A subsumption model [Bro87] [Bro89] claims intelligent behavior will emerge from low-level reactive modules. While our model includes reactive modules as part of its real-time component, the allowance for time-constrained high-level reasoning places it in the second, more traditional camp. As in a modular architecture [Fod85], we assume different, loosely coupled mechanisms for low-level reaction-perception and high-level reasoning.

Real-World Layer	Responsibilities	Layer Division	Characteristics
Application Layer	Respond to environment Generate goals Provide goal solving structure	Executive Level	agent determines next action, search executive instructs lower levels
		Intermediate Level	corrections to actions(servoing) interpret high level
		Lowest Level	survival actions, sensing, manipulating
Runtime Layer (Ephor)	Remain domain independent Use program structure representation Provide mechanisms to the application Monitor the underlying system	Interface to Application	shared data structure, mechanisms, system information to application
		Interface to real-time substrate	monitor underlying system, schedule tasks
Real-Time Substrate	Provide basic real-time properties Make more information available (e.g. resource allocation)		

Figure 2.2: The three layers in the design of a real-world application

We augment the cognitive-reactive dichotomy with an intermediate, run-time layer (Figure 2.2). All the application levels reside above the runtime layer and real-time substrate. The application interacts only with Ephor and not directly with the substrate. At the lowest [application] level are [hard] real-time periodic or aperiodic (environment responsive) tasks. These tasks generally require the same set of resources for execution to execution, and run for predicatable amounts of time.

The intermediate [application] layer serves several functions. Among them, it "catches the mistakes" of the lower level and "interprets the meaning" of the higher layer. The former involves servoing, adjusting sensors and manipulators to ensure the intended action is actually carried out by the lower level (e.g. servo robot arm to sheep). The latter involves parsing a high-level description into components (implementable code and functions) that can be understood and executed by the lower level (e.g. determine robot arm instructions to herd sheep "5").

The highest or executive level consists of planning, reasoning, information gathering and processing, decision analysis, etc.. With a traditional run-time and underlying operating system, the most significant differences between applications occur at the executive level. Unfortunately it is this level that is most task-dependent and has the fewest standard formalisms. Researchers have approached this aspect of SPARTAs from different angles. Both [GD92] and [HHP92] describe planners effective in handling varying environmental factors. Hoogeboom and Halang [HH92] propose a more general approach suggesting that "In anticipation of a deadline at which some task must be fulfilled, it should be possible to choose from different program segments the one that maintains optimum performance." We concur.

A goal in our work, and implicit in the planner specifications, is the desire to design a general architecture rather than just one for a specific application.

While there are accounts of specific applications [BT94] that have clear design principles and correct behavior, it is difficult to extract useful code from these programs to help design another SPARTA. Some other work that has looked at dynamic tradeoff decisions is Schwuttke and Gasser's [SG92] dynamic trade-off evaluation algorithm that decide which data to monitor in a spacecraft. Durfee [Dur90] suggests a more general method of supporting individual cooperating components. Other prominent work in this area of developing general mechanisms for supporting SPARTAs is by Gopinath and Schwan [GS89] who suggest objects that can move along a continuum of resource use and describe mechanisms for scheduling these objects in a distributed system.

2.3 Planner Design

2.3.1 Background

Let a *task* be something the application wishes to accomplish (e.g., save sheep) and a *technique* be a method or algorithm for accomplishing a task (e.g., planner A). Figure 2.3 illustrates the model of the program structure for a SPARTA. Throughout a program's execution it will need to execute many tasks and frequently will need to execute the same task repeatedly. If each task has only a single, sequential, fixed technique to solve it, then there will be no flexibility in choosing a technique for solving a task and thus the program will have sacrificed an entire dimension of adaptability (it can only choose different tasks indicating a different course of action).

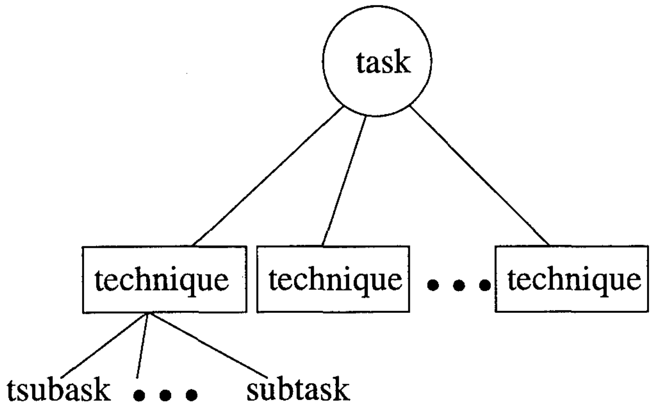

Figure 2.3: Program structure

Interacting with the real world implies coping with the unknown and the uncertain. Tasks may be generated in regards to unexpected environmental stimuli. As a specific example from our real-world shepherding application, consider the entry of a wolf into the field; a high priority task ("kill wolf") must be executed. Some tasks may take longer or shorter than expected because of a change in the environment or because of varying amounts of available resources (if while executing the "find next sheep to save" task on seven processors, six of them are preempted for other tasks, this task will take considerably longer to execute than originally expected). An unpredictable environment can also cause additional tasks to be needed while no longer requiring the results of others. For example, if while in the middle of executing the "save sheep task", the robot arm is allocated to killing a wolf, there is no reason to continue to execute the "save sheep task" since the robot arm will not be available, instead the processor(s) could be freed and given to another task. Clearly, the internal state of a SPARTA application, run-time, and operating system will be highly variable over time. To make efficient use of resources the application must cooperate with the underlying runtime to allow the whole system to adapt dynamically to varying conditions. Using worst case analysis to pre-configure the system is too inefficient [PABS91][SP94].

With our model, the SPARTA programmer can still conceptualize a program in terms of tasks that need to be executed and techniques for implementing those tasks. The programmer does not need to spend effort tracking resource allocation. The only difference between previous models for off-line applications and the model described by our planner principles for SPARTAs is the emphasis on specifying several ways of executing a given task. Of course, to take advantage of this new model, underlying support is required (see [WB93][WB96]), and the ability to inform the underlying support about the tasks and techniques. We have developed a simple scheme that allows the programmer to communicate a SPARTA's program structure to the underlying runtime environment, which we briefly describe in Section 2.4. We devote the rest of this section to discussing the details and giving examples of our planner principles.

2.3.2 Designing a Suite of Planners

It is essential for SPARTAs to maintain as much flexibility as possible both in their ability to choose different courses of action based on the environment and their ability to have multiple ways (techniques) for *determining* a particular course of action. Below we list (in order of increasing effectiveness as measured by application performance) a set of principles for planner design. After the list we describe each item in detail and provide examples. The more of the principles that are followed when designing a SPARTA planner the better the program's performance will be.

1. Provide techniques that can vary in completion time (e.g., familiar concept of anytime algorithms [LLS+91]).

2. Provide multiple techniques (to be present in Chapter 5) that:
    a) use different resources (e.g., infrared sensor/binocular vision)
    b) vary (significantly) in quantity of resources used (cpu time, etc.).
3. Provide parallel planners that:
    a) use a "bag of tasks" (processor farm) model
    b) use different resources
    c) vary in quantity of resources used.

The intent of these recommendations is to provide flexibility of resource allocation in as many dimensions as possible. The more flexibility designed at this level the more adaptable the program will be to unforeseen events since the underlying runtime environment (Ephor) will be able dynamically to select from a more diverse set of techniques and thus will more likely be able to find an appropriate technique for a given situation.

2.3.3 Description of Planner Principles

Following the outlined planner principles yields an adaptable program that allows the runtime to adjust to unexpected events and thus achieve increased performance. For each principle above we provide a detailed description and give an example from the shepherding domain described in the introduction.

1) Provide techniques that can vary in completion time

A method for meeting the challenge of time variability is to design a planner that can move along a continuum of completion times as suggested by Gopinath and Schwan [GS89]. Such a technique is similar to the motivation behind anytime algorithms or imprecise computations [LLS+91] in which after a certain minimum time the program's result improves until a final completion time. These methods allow the underlying system dynamically to allocate the maximum amount of time available to the task while still allowing for early termination if processor cycles are needed by other tasks.

An example from our shepherding application is the vision processing task of determining the centroid of each (circularly marked) sheep. A quick approximation of the centroid may be found by scanning a horizontal and then vertical line (see Chapter 6 [vKW94]). After this initial phase we have a reasonable centroid. Continuing by searching every pixel and using a weighted mean to determine the centroid will provide more accurate results, and given time, would be preferred.

2a) Provide multiple techniques that use different resources to achieve the same task

Ideally, these techniques would have non-intersecting resources, but techniques using different but not unique sets are still useful. During execution, when the high-level task implemented by these different techniques needs to be run, it will still be able to be executed even if a resource from one technique is allocated to some other task. The runtime can automatically choose to run the other technique that does not require the allocated resource.

As a specific example consider a mobile robot that has two techniques it can use to find the distance to a wall. It has an infrared sensor that may provide a fast response and a pair of binocular cameras it can also use. If both resources are free when it needs to execute the get-distance-to-wall task, then it prefers to use the infrared sensor because it is faster. However, if the high-level executive decides it is time to obtain the distance to the wall when the infrared sensors are being used to avoid an object, the runtime can still execute the task by running the technique that uses binocular vision.

2b) Provide multiple techniques that vary in quantity of resources used

These techniques will differ in the amount of resources they use and consequently the quality of the result they produce. The most straightforward example is the amount of cpu time consumed. Anytime algorithms capture this notion and are supported by Ephor, but even more significant differences yield greater adaptability, for example: emergency or reflexive $\Theta(1)$ algorithms, heuristic $\Theta(n^{2 or 3})$ algorithms, or brute force search $\Theta(2^n)$ algorithms. It is however most important to design techniques within constant factors of the expected time available for this task. Having techniques that vary widely from almost no time to lots of time is useful, but under normal situations the time available for a task is likely to be in a semi-predictable range, and providing alternatives to the runtime in that range will help increase performance. This dimension of flexibility allows the runtime dynamically to select the best technique based on the internal load on the SPARTA's resources. There may be periods of time when the application desires many tasks to be executed simultaneously and other periods of relative inactivity. We have found a diversity of planners provides the best overall behavior for a given task, because during quiet periods a higher quality technique can be run and during periods of high demand a simple technique can still be run (as opposed to being unable to run any technique).

For example, in the shepherding application we have implemented a simple planner that just looks for the first sheep it finds moving away from the center and computes the intercept to save it. We also have implemented another planner varying in the amount of lookahead performed. We implemented a depth n search (where n is the number of sheep), but in practice it never has enough time to run for $n > 4$. Lookahead is useful because, for example, it may be the case that by letting the farthest sheep from the center go and moving to the other side of the field two sheep can be saved. These different planners are extremely valuable because they provide alternatives for Ephor to choose between. We provide cursory results in Section 2.4 and perform a more thorough examination in Chapter 5 that shows how, with a variety of planners, we can achieve better behavior (more sheep confined) in the shepherding application.

3a Provide parallel planners that use a "bag of tasks" model

As mentioned earlier, SPARTAs contain parallel components that bring a new level of complexity and a new set of issues to designing real-world applications. However, this parallelism also brings new opportunities for adaptation. There are many models of parallel computation (e.g. data parallelism is natural in low-level vision). Programming a technique to have a fixed number of subtasks on a fixed set of processors is counterproductive since it does not allow for any adapting. Instead, a model of parallelism is needed that can easily and quickly change in light of varying and unpredicted environmental stimuli. Our experiments show that if the application is programmed with a "bag of tasks" model, the runtime can provide considerably better performance. In a "bag of tasks" model, work is divided up into reasonable-sized pieces and placed in a central repository. Each process removes a piece from the bag, processes it, and possibly updates shared information with the result. Examples of this model of parallelism are the Uniform System [TC88] or the Problem-Heap Paradigm [Cok91; MNS87]. This paradigm provides tremendous flexibility since the runtime can choose to run any number of processes to work on this technique.

An example from the shepherding application is a parallel planner we designed. This planner looks at the next *n* (for our experiment it was four) possible sheep saves in order to determine the best next move. Another way to cast the planning problem is to look at all permutations of the sheep in the field and count the number of sheep still confined at the end of the sequence of sheep saves and the amount of time taken to make those saves. This representation nicely fits the "bag of tasks" model since now we can place into the "bag" a set of all the possible permutations. Each (identical) process pulls a permutation out of the bag, computes the information above, and updates a central location (holding the best permutation seen so far) if it determines it has found the best option so far. We give results from applying the "bag of tasks" model to the shepherding domain in Section 2.4.

3 Provide parallel planners that: b) use different resources c) vary in quantity of resources used.

Design parallel planners that either use distinct resources or that vary greatly in the quantity of resources consumed. The arguments and benefits are analogous to those we discussed for sequential planners in **2a** and **2b**.

2.4 Supporting Results

While we present a more detailed examination of Ephor mechanisms in Chapter 5, in this section we provide a few specific results to demonstrate the effectiveness of the planner principles described in the last section. The evaluation is accomplished by using the shepherding simulator (a real-time simulator of the shepherding application), Ephor (our runtime environment), and real-time primitives from IRIX (the operating system of our 12 processor SGI Challenge). First, we briefly explain the application-Ephor interface, which is how the application informs Ephor of its tasks, techniques, and subtasks. Then, we provide results demonstrating improved performance using adaptable planner principles. We concentrate on two specific principles. We show that application behavior improves when multiple techniques accomplishing the same task are available to the runtime and when a parallel planner is used that can adapt the number of processes.

At startup, the application informs Ephor of its tasks, the different techniques it has for executing those tasks, and the specific functions (subtasks) that implement the techniques. Upon task creation, the program can specify that the task is periodic, in which case Ephor will automatically place it on the scheduling queue at the appropriate time. If the task is not periodic then the application simply informs Ephor (via a library call) when it wants that task to run (effectively the same as executing a function call). Once Ephor knows about the application's tasks, techniques, and subtasks, it dynamically selects the most appropriate technique when a given task has been requested for execution.

As a simple example from the shepherding application, Figure 2.4 shows how the vision processing task is defined. The first call `ephor_create_task` returns a handle to the vision processing task. If this task was not periodic, e.g., a task called in response to an environmental stimulus, then `ephor_periodic` would be `False`, and the program would call `ephor_run_task(vision_proc_task)` when it wanted this task run. Given the vision_proc_task handle, the application may now create as many different techniques for this task as it can find ways to perform it. For each technique the application creates all the subtasks needed to complete that technique. The subtasks are pointers to actual C functions that Ephor will run once it selects the technique to use to execute a given task.

```
vision_proc_task = ephor_create_task(ephor_periodic,
             True, ephor_priority, 1,
             ephor_task_name, "vision proc",
             ephor_rate, 16666, NULL);  /* 60 HZ */

temp_tech = ephor_create_technique(vision_proc_task,
             ephor_cpu_time, 5000, NULL);

vision_proc_id = ephor_create_task(vision_proc_task,
             temp_tech, ephor_imp_function,
             vision_processor, NULL);
```

Figure 2.4: Application-Ephor interface

2.4.1 Evaluation of a Suite of Techniques

The first planner principle we evaluate is the usefulness of generating different techniques for accomplishing the same task. Here we will examine the performance of two planners (planner A and B) – a more thorough analysis of these tradeoffs may be found in Chapter 5. Both of these planners figure out the next sheep to save but differ in how long they take to run and how many sheep are contained (in steady state) when running on an unloaded cpu. To guarantee accurate measurement (no competing load) for this experiment, we dedicate one processor to the planner function. A sheep can travel from the center to the edge of the table in 10 seconds and the shepherd can travel this distance in about 1/3 of a second.

Planner A computes a list of all the sheep moving away from the table center that the shepherd has time to reach, sorted by distance from the center. It then determines the best order for saving the next four sheep: this requires future prediction of sheep movements. The best sequence is the one maximizing the number of sheep saved. Among the orderings that save equal number of sheep, preference is given to the ordering taking less time. The first sheep in the sequence is saved and the planner starts over. Planner A performs the best under no load but takes the longer time to run (about one second). Planner B is a reactive planner (no look-ahead strategy) that simply tries to save the sheep farthest from the center. It runs much faster than A (about 8 milliseconds) but does not perform nearly as well under no load: if by letting the farthest sheep go, it is possible to save the next two and otherwise not, A will save the two sheep while B will save only one.

To compare the different planners under simulated conditions of parallel activity in other parts of the SPARTA, a controlled load was placed on the processor that the planner was running on. Since we were using a multiprocessor we could

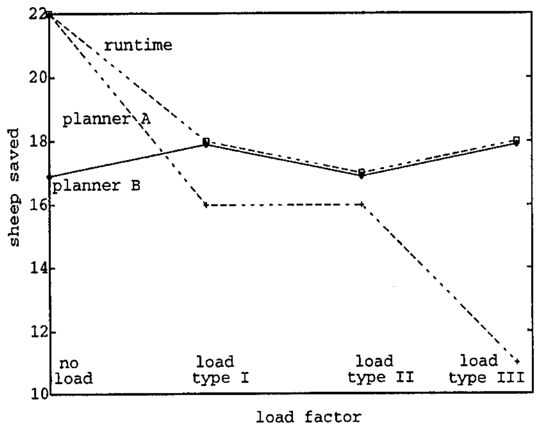

Figure 2.5: Adapting to high fixed loads

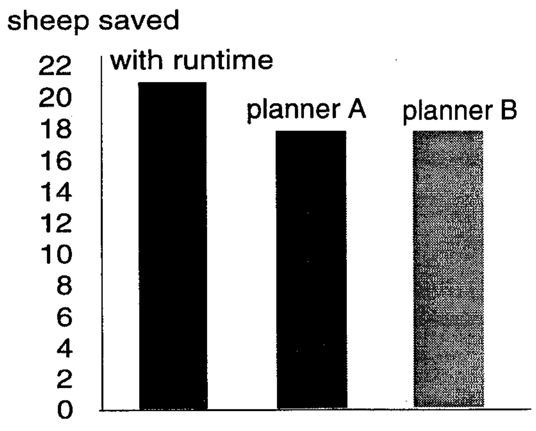

Figure 2.6: Adapting to high variable loads

vary the load experienced by the planner process without affecting any of the other processes in the system. We also had tight control over how much load was experienced on the processor running the planners. The line graph of Figure 2.5 (on the left) gives performance for a set of fixed loads (the load does not vary throughout the entire execution - an unrealistic model since in a real application tasks will come and go, but it allows us to see the comparative benefits of each of the planners), while the bar chart of Figure 2.6 gives average performance when the cpu load varies during the run (like actual program execution).

The experiment (Figures 2.5 and 2.6) demonstrates the effect of a high load. Recall that B runs about 40 times faster than A. Planner A is expected to outperform B with no load, but under increased load planner A might not complete its calculations in time, thus planner B is expected to outperform A under high load. The loads are plotted on a logarithmic scale: **load type II** is twice as much background load as **load type I** and half as much as **load type III**. Indeed there is a dramatic decrease in performance of planner A under higher loads while planner B remains fairly constant. In a fixed load environment the run-time can select the better of the two planners, thus achieving the best performance in all cases. In the second half of the experiment the load varied through time. Half the time there was no load and half the time there was load. When there was a load it was divided evenly (by thirds) amongst the different load types. Figure 2.6 represents how A, B, and the run-time mixture perform under varying load; best performance occurs when Ephor dynamically selects the planner to suit the (currently) available resources. It is clear that dynamically selecting between planners improves application performance.

2.4.2 Evaluating Adaptable Parallel Planners

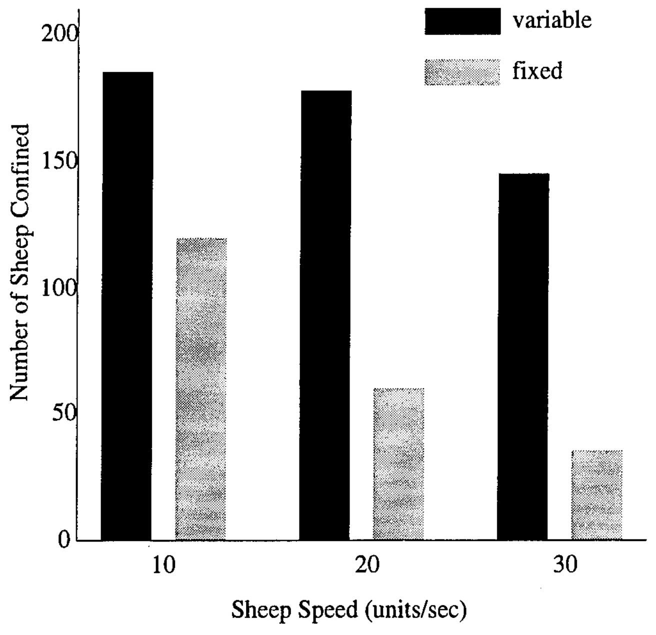

Figure 2.7: Fixed versus adaptable parallel planners

The second planner principle we evaluate is the "bag of tasks" (see **3a**) model for parallel planner design. As we have mentioned, a key aspect of performing well in the real world is being able to adapt. This adaptability applies both to the runtime and the application. Our principle of using a "bag of tasks" model is motivated by the fact that it provides considerable flexibility when considering the amount of processing power to allocate to a planner.

In this experiment we again assume a model of varying load as would be observed in a real application. The parallel planner has been written using a "bag of tasks" model. The planner looks four saves into the future. The possible permutations for the next four sheep saves are placed into a central queue where they are removed by as many processes as the runtime has decided to invoke in this particular task instantiation. This planner is qualitatively different from the planners discussed in Section 2.4.1 and the environment is quantitatively different, so the results should not be compared.

The results appear in Figure 2.7. The load of the application allowed for between 1 and 7 processors to be available to the save-sheep task at different points throughout its execution. The "fixed" bar represents the application's behavior assuming it could not adapt and adjust to use the extra processors available at various points throughout the execution. The variable bar represents when Ephor was allowed dynamically to allocate more processors for the parallel planner when they were available (see Chapter 4 for details on the dynamic parallel process control mechanism). We vary the response time expected from the planner by varying the rate the sheep move. Notice that when the response demands placed on the application increase, it becomes more important for the planner to have been designed allowing for differing number of processes to be used. This figure illustrates the benefit of using a *flexible* parallel planner. When the sheep are moving around quickly, the adaptable parallel planner can confine over three times as many sheep as the fixed (one process) parallel planner. The important aspect of this discussion is not that parallel planners (versus sequential ones) can improve the performance of applications, rather, that in real-world applications we need parallel planners that can dynamically vary the number of processors they use. Equally important, we need a runtime such as Ephor, that can support this desired behavior.

This graph represents the interesting part of the state space for the save-sheep planner. If the sheep move very slowly (1-5 units/sec) then either planner will have enough time to do something reasonable, and similarly if the sheep are moving extremely quickly (greater than 60 units/sec) neither planner will have enough time to do anything. Achieving perfect speedup when running on seven processors (versus one) would allow the planner to run seven times as fast. This does not however, translate into a seven-fold improvement in application behavior. In fact there may be some programs that do not benefit or benefit very little from decreased running time. However, many real-world applications will be able to benefit from being able to perform more computation in less time. Exploiting the parallel dimension of programming a SPARTA can substantially improve application behavior.

2.5 Runtime Support

In the last two sections we argued and provided results showing that by building flexibility into the application layer better performing SPARTAs could be achieved. This flexibility was achieved by (among other possibilities) having multiple techniques for solving a given task and by having parallel planners that could take advantage of additional processors. While it is conceivable that building such capability into the application would be possible, there are many benefits to designing a general purpose runtime, such as Ephor, to provide the required support. These benefits include elimination of replicated work across applications, better performance through careful study of possible mechanisms, and improved modularity. In the rest of this chapter we will further motivate the need for a runtime between the application layer and the kernel, provide a model for such a runtime, and describe some of the pertinent implementation issues involved in constructing the runtime.

2.5.1 Motivating Factors

Our research work in this area developed from a desire to provide systems support for a class of real-world and AI applications. We devised the shepherding problem to embody many of the properties of the applications we wished to support. A brief overview of shepherding was provided earlier in this chapter and a complete description appears in Chapter 6. Briefly, *sheep* (small vehicles) move around in a *field* (table) and a *shepherd* (robot arm) tries to maximize the number contained in the field. Before and during implementation we observed that there were several mechanisms that would have been useful but were not provided by current real-time systems such as concurrency control, dynamic technique selection, help in priority assignment, etc. As Stankovic [Sta92] notes, "Because of these reasons many researchers believe that current kernel features provide no direct support for executing difficult time problems, and would rather see more sophisticated kernels..." A system that provided these features would be very useful. Rather than including these features into the kernel and sacrificing kernel predictability, we placed the additional functionality into Ephor, our runtime environment. This allows the designer to use the real-time kernel most appropriate for their environment. The application still receives the same functionality (actually more) and we maintain a predictable kernel.

2.5.2 SPARTA Properties

In designing Ephor in conjunction with the shepherding application we wanted to ensure the mechanisms developed for shepherding would be applicable to other programs. To do so, we designed the shepherding application to have many of the same properties as the soft real-time applications mentioned in the introduction. These applications contain an element of search whereby the agent determines the next course of action. Most are designed around a high-level executive instructing lower levels. The executive reasons using a model of the real world and carries out actions in it. The world is governed by general principles, but is not predictable. There is often an intermediate layer responsible for small corrections to the requested action (servoing). Also, there is often a low-level layer whose actions need to be carried out constantly and can occur "subconsciously", i.e., without intervention from the higher levels.

2.5.3 SPARTA Design Problems

Under the standard taxonomy there is only an application and kernel. Any operations or functions not performed by the kernel are the responsibility of the application. While the application needs to respond to the environment, determine the next course of action, and evaluate current progress, to perform reasonably it also needs to:

1. Monitor the load of the underlying processors. Often the application has a choice of tasks to accomplish the same objective. Rather than submit impractical tasks, if the application knew the amount of resources available, it could quickly choose the appropriate task to submit.
2. Maintain a list of allocated resources. As in 1, a wiser task submission based on task resource allocation yields better behavior.
3. Track execution times of tasks, especially highly variable ones. We have found that locally the execution time of a task is fairly predictable, so knowing the last several execution times is useful (see Chapter 4).
4. Determine the correct interleaving of prioritized tasks.

It was our objective in designing Ephor to ensure these enumerated ideas could be handled by the generalized runtime, thus removing a significant burden from the SPARTA programmer.

2.5.4 Ephor Paradigm

Designing and implementing the shepherding application led us to the conclusion that there is considerable functionality above the intended realm of real-time kernels that an increasingly large class of real-time applications strongly desire. It is best not to remove the predictability of the kernel since many critical applications depend on this property. However, there is a large class of applications willing to relax the tight constraints to gain increased functionality. These soft real-time applications will sacrifice predictability with or without a runtime. Further, if Ephor's mechanisms are not desired for a portion of the application they may be ignored causing no overhead to that portion. A runtime layer, such as Ephor, will have a positive effect on SPARTA design. The only penalty if none of Ephor's mechanisms are used (or used only very minimally) is the one processor normally reserved to run it will be unavailable for other tasks. However this scenario is unlikely as Ephor provides a wide range of mechanisms and any SPARTA should be able to take advantage of a healthy set of mechanisms. In the next section we describe the responsibilities of each layer under the Ephor paradigm.

2.6 Layer Responsibilities

By defining the responsibilities of each layer, we clearly define the obligations of the SPARTA designer. More importantly, we can specify the interface to the

Real-World Layer	Responsibilities	Layer Division	Characteristics
Application Layer	Respond to environment Generate goals Provide goal solving structure	Executive Level	agent determines next action, search executive instructs lower levels
		Intermediate Level	corrections to actions(servoing) interpret high level
		Lowest Level	survival actions, sensing, manipulating
Runtime Layer (Ephor)	Remain domain independent Use program structure representation Provide mechanisms to the application Monitor the underlying system	Interface to Application	shared data structure, mechanisms, system information to application
		Interface to real-time substrate	monitor underlying system, schedule tasks
Real-Time Substrate	Provide basic real-time properties Make more information available (e.g. resource allocation)		

Figure 2.8: The three layers in the design of a real-world application

runtime, treating it as a "black box" with respect to the SPARTA programmer. A contribution and important part of our work is a clean separation of responsibilities for each layer and a description of mechanisms provided by Ephor independent of how they are coded. Thus, we not only have provided mechanisms, but a methodology.

Figure 2.8 (repeated from Figure 2.2 for convenience) shows the three layers in designing a SPARTA and underlying system. Information is shared across the runtime-application boundary by the data access functions. For example, ephor_get_task_time (my_task) will return the execution time for my_task. These access functions provide information to flow both ways. The user application can request information that Ephor has been gathering, or the application can override a selection among different techniques by instructing Ephor to use the specified one. The clean breakdown of information communication in Fig. 2.9 has been achieved by dividing layer responsibilities as detailed below.

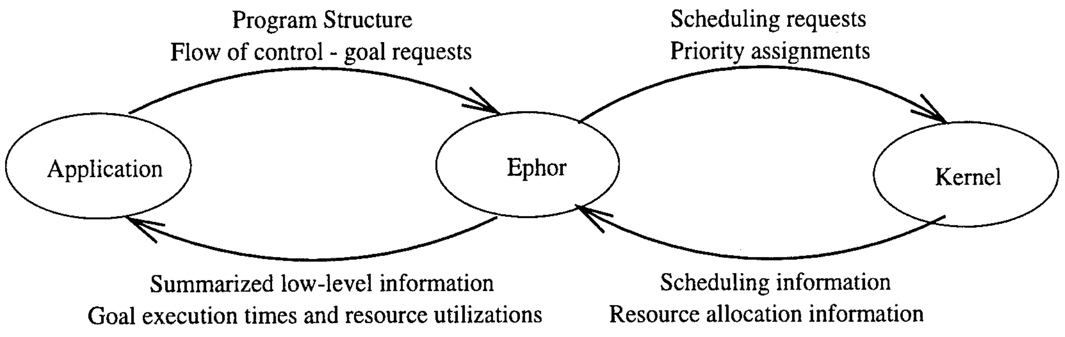
Program Structure
Flow of control - goal requests
Scheduling requests
Priority assignments
Application
Ephor
Kernel
Summarized low-level information
Goal execution times and resource utilizations
Scheduling information
Resource allocation information

Figure 2.9: Layer information exchange

2.6.1 Application

The application layer is solely responsible for responding to the environment. The application is responsible for communicating to the runtime the different tasks it will run throughout its execution, the different techniques it has for executing the tasks, and the relative benefit (simply a linear order of preference) of each technique. The application is responsible for determining the interaction of the environment and tasks to produce the intended behavior. The application is responsible for indicating when a new task needs to be executed in response to an environmental stimulus, or as the result of a previously completed task. In essence the application must provide the flow of control to produce the desired program.

2.6.2 Runtime

The runtime receives the structure of the tasks the application will submit throughout its execution via the initialization functions. As we mentioned in Section 2.3, there can (should) be different techniques for each task. Each technique may have one or more subtasks that actually implement the technique. The runtime is responsible for determining the execution time of the different techniques for executing the tasks. The application can explicitly provide the times (required for worst case scheduling), or the runtime may dynamically gather them during the program's execution. The latter method allows Ephor to adapt in a changing environment. Ephor is responsible for determining and running the appropriate items needed for completing a selected technique. In selecting the technique to execute a requested task, Ephor needs to be aware of the program structure and application and the internal state of the system. The runtime is therefore responsible for interfacing to the underlying kernel to obtain the information it needs. It is responsible for monitoring the following resources necessary to select dynamically the best technique: processor load, expected available processors for running parallel techniques, and other current resource allocation such as range sensors, manipulators, or cameras. Ephor is responsible for maintaining a central location where resource allocation information can be quickly and coherently obtained by either Ephor or the application.

To provide a reasonably comprehensive runtime package we have implemented many mechanisms in Ephor. They will be discussed in detail in Chapter 5, here we give a list to provide an overview of Ephor's functionality:

1. Dynamically select technique based on internal system state.

2. Schedule tasks using derivative worst case or adaptive scheduling policies.

3. Dynamically control the placement and quantity of parallel processes.

4. De-schedule all running subtasks associated with a particular task.

5. Automatically time tasks and update their status block.

6. Provide access functions to share information between the runtime and application.
7. Detect and recover from overdemand.
8. Allow early termination of tasks.
9. Automatically allocate resources based on task priority.
10. Provide user-conscious synchronization.

In summary, Ephor is responsible for monitoring the underlying system, summarizing the internal system information for the application layer, and dynamically selecting the appropriate techniques to execute the requested tasks.

2.6.3 Kernel

We assume a hard real-time substrate. The kernel is expected to provide fundamental real-time properties and mechanisms such as a predictable scheduling policy, a real-time clock, guaranteed deadlines, task priorities, and other properties typically associated with real-time kernels [Sta92]. The kernel is responsible for handling interrupts, and allocating resources as directed by Ephor.

2.7 Implementation

There are three primary components to Ephor or any runtime layer that resides between the application and the kernel. There is the interface to the application, the interface to the kernel, and the code for implementing the functionality of the runtime. In this section we describe the implementation of these portions of Ephor, both to give a general feel for what would be required in a generic runtime layer and to provide background for results based on Ephor presented in Chapters 4 and 5.

Ephor needs to interface with the application layer. It needs to receive the program structure (in terms of tasks, techniques, and their subtasks) and details about that structure, e.g., worst case time for a task. It also makes available information about the system to the application layer, e.g., most recent execution time of a task. From the application's perspective, this information is accessed via Ephor library functions. This approach was taken rather than providing direct access to the data structure since it allows Ephor to prevent the application from setting inappropriate values. It still allows asynchronous access to the data since the function is actually run by the caller and Ephor does not respond directly to the request.

Ephor also needs to interface with the underlying kernel. Since we did not desire the additional difficulties of modifying kernel code, our implementation of the Ephor - kernel differs slightly from a production implementation and was sufficient to test and demonstrate our ideas in a prototype. At initialization Ephor creates one process for every task that may be executed throughout the application's existence. This is a common technique employed in real-time systems to avoid the highly variable time of creating a process and reduce it to the more predictable job of performing a context switch (IRIX guarantees a 200 μs bound for this). Since Ephor does not have direct control of the processes a footer and header is inserted into each task. Ephor and the process then cooperate to achieve the desired behavior. For example, if Ephor chose to run a task on processor 7, it sets the migrate_processor flag and indicates processor 7. This flag is checked in the header of each task and if high then the task calls an IRIX primitive to migrate to the desired processor.

There are four actions taken in the header[2] of a task. The first action is to block on a semaphore and wait for the user-level scheduler in Ephor to unblock the task. The header also checks a flag to see if Ephor has migrated or changed the priority of the task. If so, it calls the appropriate IRIX primitive to execute the action. Finally, the header starts the 21 ns resolution timer. The footer of the task simply stops the timer and updates the time fields. The need for the cooperation between Ephor and the process occurs because we did not make kernel modifications. In a production system, the above items would be handled by the kernel just before running a task. The task's model then is to sit in an infinite loop starting with a P on a semaphore (that Ephor will V). Upon being unblocked, the (header of the) task checks a migrate flag, sets the appropriate priority, and starts a fine granularity clock. This is all hidden to the user; the only responsibility of the user is to call ephor_begin_proc at the beginning of a function for running a subtask and call ephor_end_proc at the end.

The scheduler is described in greater detail in Chapter 4 and is based on a parallel version of rate monotonic scheduling. The scheduler sorts the tasks first by priority and then by period. It then starts with the first available processor and proceeds through the list placing the tasks on to the lowest number number possible while still meeting deadline requirements. The user level dispatcher for Ephor executes the decisions made by the scheduler. It runs the tasks by performing a V (remember each task was blocked on a semaphore). Since Ephor tracks the resources required by each task (cpu use and physical resource use) when the task is run by the kernel it will not block waiting for physical resources nor will it be rejected by the real-time kernel for lack of schedulability (it is still possible to miss deadlines because of tasks over running). Thus, Ephor, the runtime layer, is acting as the appropriate intermediary between the application layer and the

[2]Section 6 contains the C code for the header and footer

kernel, tracking resource allocation, and insuring good use of the machine.

2.8 Summary

In this chapter we provided a model of a SPARTA. We made a set of recommendations for how to design a SPARTA, which motivated the desire to have underlying support. We motivated and described the benefit of having an intermediate layer such as Ephor to provide this support, and briefly presented results indicating the Ephor can improve performance in addition to providing the support desired by SPARTAs. In following chapters we will more carefully examine each of the mechanisms in Ephor.

3 Synchronization in the Presence of Multiprogramming

Multiprogramming occurs when there are more processes than processors. It can be implemented in many different ways but the outcome is always that some processes will have to share a CPU. This implies that they will not always be executing. The process may be swapped out at the end of its quantum, or in a real-time system when a higher priority task enters the system. Either way, if the process is involved in a synchronization operation performance can suffer. Ephor is designed to facilitate sharing of information both ways across the kernel - application interface. Using the same philosophy of sharing information, we have developed a library of synchronization mechanisms that combat the synchronization difficulties encountered due to multiprogramming. Although not as extensive as Ephor, the library provides a set of synchronization mechanisms that interact with the kernel and provide improved performance.

In Chapter 5 we describe a planner that is part of the shepherding application. The high performance of the planner relies on the fact that processes are able to come and go throughout the execution of the shepherding application. The planner occasionally accesses a central data structure that requires a mutual exclusion lock. In such an environment it is imperative to use a synchronization mechanism capable of handling the possibility of preemption. Other synchronization researchers have also recognized the importance of combining the ability to handle preemption and real-time. Both Craig [Cra93] and Takada and Sakamura [TS94] have examined the issue of designing locks for real-time systems. In this chapter we will describe many synchronization mechanisms, some that are suitable for real-time systems. We apply our ideas of sharing information to develop these synchronization mechanisms for *both* small and large scale machines.

3.1 Introduction

Traditionally, synchronization algorithms have been designed to run on a dedicated machine, with one application process per processor, and can suffer serious performance degradation in the presence of multiprogramming. Problems arise when running processes block or, worse, busy-wait for action on the part of a process that the scheduler has chosen not to run. We show that these problems are particularly severe for scalable synchronization algorithms based on distributed data structures. We then describe and evaluate a set of algorithms that perform well in the presence of multiprogramming while maintaining good performance on dedicated machines. We consider both large and small machines, with a particular focus on scalability, and examine mutual-exclusion locks, reader-writer locks, and barriers. The algorithms we study fall into two classes: *preemption-safe* techniques that decide whether to spin or block on the basis of heuristics, and *scheduler-conscious* techniques that use information from the scheduler to drive more accurate decisions. We show that while in some cases either method is sufficient, in general, sharing information across the kernel-user interface both eases the design of synchronization algorithms and improves their performance.

Mutual exclusion locks are very common in SPARTAs. Search algorithms, planners, and blackboards are all used in SPARTAs and have central data structures that need to be protected. In Chapter 2 we describe a parallel planner that is part of the shepherding and that requires a central lock. Any system employing the blackboard approach [EL75; EHRLR80] will also need mutual exclusion locks. Although lock use in SPARTAs is most common, other forms of synchronization are sometimes required. For example, potential field control methods have been popular in trajectory planning and navigation control for a decade. Computing these fields is done by SOR or Gauss-Seidel iterative computation. More sophisticated methods such as boundary element analysis have been suggested for real-time use, and have the same computational flavor. Particular recent examples include Gans's work [Gan96] on vehicle following using potential fields, as well as the robot navigation work by Grupen and Connolly [GCSB95; CG93] on which Gans's work is based. In a SPARTA there are many tasks, and although often the application has the whole machine, each task is preempted as in a multiprogramming system to let other or higher priority tasks execute. It is especially important in a real-time environment that scheduler-conscious synchronization algorithms are used. They provide both the predictability and improved performance needed in these environments.

One of the most basic questions for any synchronization mechanism is whether a process that is unable to continue should *spin*—repeatedly testing the desired condition—or *block*—yielding the processor to another, runnable process. Spinning makes sense when the expected wait time of a synchronization operation is less than twice the context switch time, or when the spinning processor has nothing else useful to do. Researchers have developed a wealth of busy-wait (spinning) mechanisms, including mutual exclusion locks, reader-writer locks (which allow concurrent access among readers, but guarantee exclusive access by writers), and *barriers* (which guarantee that no process continues past a given point in a computation until all other processes have reached that point). Of particular interest in recent years have been *scalable* synchronization algorithms, which employ backoff or distributed data structures to minimize contention [And90; GT90; HFM88; KSU93; Lee90; Lub89; MLH94; MCS91a; MCS91b; MCS91c; SMC94; YA93; YTL87].

Unfortunately, busy-waiting in user-level code tends to work well only if each process runs on a separate physical processor. If the total number of processes in the system exceeds the number of processors, them some processors will have to be multiprogrammed. The unexpected nature of scheduling decisions cause problems for synchronization algorithms and can seriously degrade performance when:

- a process is preempted while holding a lock,
- a process is preempted while waiting for a lock and then is handed the lock while still preempted, or
- a process spins when some preempted process could be making better use of the processor.

In our experiments, we have found that algorithms that provide excellent performance in the absence of multiprogramming may perform orders of magnitude worse when multiprogramming is introduced. These results suggest the need for *scheduler-conscious* synchronization—techniques that share information with the scheduler in order to avoid bad interactions.

Several research groups have addressed one or more aspects of scheduler-conscious synchronization. Some have shown how to avoid preempting a process that holds a `test_and_set` lock, or to recover from this preemption if it occurs. Others have developed heuristics that allow a process to guess whether it would be better to relinquish the processor, rather than spin, while waiting for a lock. We refer to these techniques as *preemption-safe*—with only a small amount of help from the scheduler, they avoid spinning for unbounded periods for events that will not happen.

In this chapter we provide a relatively comprehensive treatment of preemption-safe synchronization and of scheduler-conscious techniques that exploit additional information from the scheduler. We cover mutual exclusion locks, reader-writer locks, and barriers, for both large and small machines. Our contributions include:

- a preemption-safe ticket lock and scheduler-conscious queue lock, both of which provide FIFO servicing of requests and scale well to large machines;
- a fair, scalable, scheduler-conscious reader-writer lock (the non-scalable version is trivial);
- a scheduler-conscious barrier for small machines (in which a centralized data structure does not suffer from undue contention, and in which processes can migrate between processors); and
- a scheduler-conscious barrier for large machines that are partitioned among applications, and on which processes migrate only when repartitioning.

Our preemption-safe locks employ a simple extension to the kernel-user interface that allows a process, within limits, to control the points at which it may be preempted. Our scheduler-conscious algorithms (both locks and barriers) employ additional extensions that inform an application about the status of its processes and the processors on which they run. Preemption-safe techniques employ heuristics to guess this information.[1] The heuristics achieve acceptable performance in certain cases, but in general we find that use of a wider kernel interface results in cleaner code and better performance.

We assume the availability of special instructions that allow a process to read, modify, and write a shared variable as a single atomic operation.[2] Examples include `test_and_set`, `fetch_and_increment`, `fetch_and_store`, and `compare_and_swap`, The atomic instructions used in our algorithms can all be emulated efficiently by `load-linked` and `store-conditional`.

We emphasize large machines not only because scalable algorithms are newer and hence less studied, but also because scheduler-conscious synchronization is inherently *harder* for scalable algorithms based on distributed data structures. Scalable algorithms have difficulties because their deterministic ordering of processes can conflict with the actions of the scheduler in a multiprogrammed system. For example: a mutual exclusion lock may keep waiting processes in a FIFO queue, either for the sake of fairness or to minimize contention. The algorithm's performance is then vulnerable not only to preemption of the process in the critical

[1]The distinction between preemption-safe and scheduler-conscious algorithms is not sharp. Scheduler Activations [ABLL92], for example, allow an application to completely and accurately track the status of its processes and processors. It does not share information across the kernel-user interface, but might nonetheless be classified as scheduler-conscious.

[2]Some multiprocessors, especially the larger ones, provide more sophisticated hardware support for synchronization. Examples include the queue-based locks of the Stanford Dash machine [LLG+92], the QOLB (queue-on-lock-bit) operation of the IEEE Scalable Coherent Interface [JLGS90], and the near-constant-time barriers of the Thinking Machines CM-5 and the Cray Research T3D. It is not yet clear whether the advantages of such special operations over simpler read-modify-write instructions are worth the implementation cost.

section, but also to preemption of processes near the head of the waiting list—the algorithm may give the lock to a process that is not running [ZLE91]. Similarly, a barrier algorithm may keep processes in a tree, in order to replace $O(n)$ serialized operations on a counter with $O(\log n)$ operations on the longest path in the tree. But then processes must execute their portions of the barrier algorithm in the order imposed by the tree. If the processes on a given processor are scheduled in a different order, and if they simply yield the processor when unable to proceed (as opposed to waiting on a kernel-provided synchronization queue), then the scheduler may need to cycle through most of the ready list *several times* in order to achieve a barrier [ML91].

The rest of the chapter is organized as follows. Section 3.2 discusses related work. It explains in more detail why synchronization algorithms suffer under multiprogramming, why scalable synchronization algorithms are particularly susceptible to multiprogramming effects, and why previous research does not fully remedy the problem. Section 3.3 describes our kernel interface and compares it to alternative approaches, such as process-to-process handshaking and experience-based heuristics, that use a more conventional interface. Section 3.4 describes our preemption-safe and scheduler-conscious algorithms. Section 3.5 describes our experimental environment and presents performance results. Conclusions appear in Section 3.6.

3.2 Background

3.2.1 Preemption-Safe Small-Scale Locks

It is widely recognized that lock-based algorithms (i.e. mutual exclusion and reader-writer locks) can suffer performance losses when a process is preempted while in a critical section. Remaining processes cannot access the shared data structure or protected resource until the preempted process releases the lock it is holding.

Ousterhout [Ous82] introduced *spin-then-block* locks that attempt to minimize the impact of preemption (or other sources of delay) in critical sections by having a waiting process spin for a small amount of time and then, if unsuccessful, block. Karlin et al. [KLMO91] present and evaluate a richer set of spin-then-block alternatives, including *competitive* techniques that adjust the spin time based on past experience.[3] Their goal is to adapt to variability in the length of critical sections, rather than to cope with preemption. Competitive spinning works best when the behavior of a lock does not change rapidly with time, so that past behavior is an appropriate indicator of future behavior.

[3]A competitive algorithm is one whose worst-case performance is provably within a constant factor of optimal worst-case performance.

Zahorjan et al. [ZLE88; ZLE91] present a formal model of spin-wait times. For lock-based applications in which all processes on a given processor belong to the same application, they show that performance problems can be avoided if the operating system simply partitions processes among processors and allows the application to make intra-processor scheduling decisions (never preempting a process with a lock).

Several groups have proposed extensions to the kernel/user interface that allow a system to avoid adverse scheduler/lock interactions while still doing scheduling in the kernel. The Scheduler Activation proposal of Anderson et al. [ABLL92] allows a parallel application to recover from untimely preemption. When a processor is taken away from an application, another processor in the same application is given a software interrupt, informing it of the preemption. The second processor can then perform a context switch to the preempted process if desired, e.g. to push it through its critical section. In a similar vein, Black's work on Mach [Bla90] allows a process to suggest to the scheduler that it be de-scheduled in favor of some specific other process, e.g. the holder of a desired lock. Both of these proposals assume that process migration is relatively cheap.

Rather than recover from untimely preemption, the Symunix system of Edler et al. [ELS88] and the Psyche system of Marsh et al. [MSLM91] provide mechanisms to avoid or prevent it. The Symunix scheduler allows a process to request that it not be preempted during a critical section, and will honor that request, within reason. The Psyche scheduler provides a "two-minute warning" that allows a process to estimate whether it has enough time remaining in its quantum to complete a critical section. If time is insufficient, the process can yield its processor voluntarily, rather than start something that it may not be able to finish.

3.2.2 Scalable Locks

Centralized locks can create substantial amounts of contention for memory and for the processor-memory interconnect. The key to good performance is to minimize active sharing. One option is to use backoff techniques [And90; MCS91a] in which a processor that attempts unsuccessfully to acquire a lock waits for a period of time before trying again. The amount of time depends on the estimated level of contention. Bounded exponential backoff works well for `test_and_set` locks. Backoff proportional to the number of predecessors works well for ticket locks.

A second option for scalable locks is to use distributed data structures to ensure that no two processes spin on the same location. The queue-based spin locks of Anderson [And90] and of Graunke and Thakkar [GT90] minimize active sharing on coherently-cached machines by arranging for every waiting processor to spin on a different element of an array. Each element of the array lies in a separate, dynamically-chosen cache line, which migrates to the spinning processor. The queue-based spin lock of Mellor-Crummey and Scott [MCS91a] represents its queue with a distributed linked list instead of an array. Each waiting processor uses a `fetch_and_store` operation to obtain the address of the list element (if any) associated with the previous holder of the lock. It then modifies that list element to contain a pointer to its *own* element, on which it then spins. Because it spins on a location of its own choosing, a process can arrange for that location to lie in local memory even on machines without coherent caches. Magnussen et al. [MLH94] have shown how to modify list-based queue locks to minimize interprocessor communication on a coherently-cached machine. Others have shown how to build queue-based scalable reader-writer locks[KSU93; MCS91b].

In order to have every process spin on a separate variable, queue locks require that processes acquire the lock in a deterministic (generally FIFO) order. If a process is preempted while awaiting its turn to access a shared data structure, processes later in the order cannot proceed even if the lock is released by the original owner—the lock will be passed to the preempted process instead. This problem was noted by Zahorjan et al. [ZLE91], but no solution was suggested. In Section 3.4 we present two mutual exclusion locks and a reader-writer lock that solve the problem by bypassing preempted processes in the queue and having them retry for the lock when they resume execution.[4]

3.2.3 Alternative Approaches to Atomic Update

Alternatives to the use of preemption-safe or scheduler-conscious locks include *lock-free* and *wait-free* data structures and remote object invocation.

Herlihy [Her91; Her93] has led the development of *lock-free* and *wait-free* data structures. The algorithms are designed in such a way as to guarantee both atomicity and forward progress, despite arbitrary delays on the part of individual processes. The key idea in most of these algorithms is to modify a copy of (a portion of) the data structure, and then swap it for the original in one atomic step (assuming the original has not been modified since the copy was created). Tolerance of arbitrary delays means that lock-free and wait-free data structures are immune to the performance effects of inopportune preemption. It also means that they can tolerate some page faults and even certain kinds of hardware failure, something none of the techniques in the previous section can do. Unfortunately, the current state of the art in general-purpose lock-free and wait-free synchroniza-

[4]The mutual exclusion locks originally appeared in a conference publication [WKS94]. Extensions for real-time systems have appeared in subsequent papers by others [Cra93; TS94].

tion techniques incurs substantial performance overhead, even when there is no competition for access to the data structure.

A second way to avoid the use of locks is to create a manager process that is responsible for all operations on the "shared" data structure, and to require other processes to send messages to the manager. This sort of organization is common in distributed systems. It can be cast as a natural interpretation of monitors, or as *function shipping* [LHM$^+$84; SG90] to a common destination. In recent years, several machines have been developed that provide hardware support for very fast invocation of functions on remote processors [KJA$^+$93; NWD93]. Even on more conventional hardware, programming techniques such as *active messages* [vCGS92] can make remote execution very fast. Because computation is centralized and requests are processed serially, active messages provide implicit synchronization. On the other hand, they do not permit concurrency and can only be used when the manager is not a bottleneck.

3.2.4 Barriers

In some SPARTAs barriers form a key component structuring the parallel computation. In this chapter we study the SOR application to evaluate the different barrier algorithms. In a typical SPARTA application, SOR is one way to implement the potential field computation needed by Gans [Gan96] for vehicle following control. Potential field algorithms have been used in many real-time robot applications and hardware has been designed for them because they are so computation intensive [GCSB95; CG93; CG95; RK89; TB91; Gan96]. More efficient and sophisticated algorithms such as boundary element methods have the same computational flavor. In real-time applications it is especially important to use scheduler-conscious algorithms. The optimal bus-based algorithm we describe in this section is ideally suited for the real-time robotic applications based on potential field computation because it allows the number or processes to vary as recommended in Chapter 2 yet still guarantees that processes do not wait (for peers) unnecessarily and always yield the processor rather than spin for an unpredictable amount of time.

Barrier synchronization algorithms force processes to wait at a specified point in the computation until all their peers have arrived at that same point. From a scheduling point of view, the principal difference between locks and barriers is that while the time between lock acquisition and release is generally bounded and short (one critical section's worth of computation), the time between consecutive barriers can be arbitrarily long. This means, for example, that while it may be acceptable to disable preemption in a process that holds a lock, it is not acceptable to do so in a process that must continue to execute in order to reach the next barrier.

Performance loss in barriers occurs when processes spin uselessly, waiting for preempted peers. When a process on a multiprogrammed processor spins at a barrier that has not yet been reached by some other process that could be making use of the processor, it may waste as much as a quantum, reducing system throughput and likely increasing the computation's critical path length.

Inspired by Karlin et al., we have developed small-scale, centralized barriers in which a process attempts to guess whether it is running in a multiprogrammed environment based on how long it had to wait during previous barrier episodes [KW93]—if it thinks it is being multiprogrammed it blocks; otherwise it spins. This heuristic ensures competitive performance, but a small extension to the kernel/user interface allows a process to make an *optimal* decision, blocking if and only if its processor could be given to another process that has not yet reached the barrier. We present both the heuristics and the optimal technique in Section 3.4.3.

Centralized barriers are generally based on counters, and pose two obstacles to scalable performance on large machines. First, as with locks, simultaneous attempts to access the counter can lead to unacceptable levels of contention. Second, even in the absence of contention, serial counter updates imply an asymptotic running time of $O(p)$, which becomes unacceptable as the number of processors p grows large. Several researchers have shown how to solve these problems by building barriers based on log-depth tree- or FFT-like patterns of point-to-point notifications among processes [AJ89; HFM88; Lee90; Lub89; MCS91a; MCS91c; SMC94; YTL87]

Unfortunately, the deterministic notification patterns of scalable barriers may require that processes run in a different order from the one chosen by the scheduler. The problem is related to, but more severe than, the preemption-while-waiting problem in FIFO locks. With a lock the scheduler may need to cycle through the entire ready list before reaching the process that is able to make progress. With a scalable busy-wait barrier, Markatos et al. have shown [ML91] that the scheduler may need to cycle through the entire ready list a logarithmic number of times (with a full quantum's worth of spinning between context switches) in order to achieve the barrier. To avoid this problem, they suggest (without an implementation) that blocking synchronization be used among the processes on a given processor, with a scalable busy-wait barrier among processors. (Such *combination* barriers were originally suggested by Axelrod [Axe86] to minimize resource needs in barriers constructed from OS-provided locks.) The challenge for a combination barrier is to communicate partition information from the scheduler to the application, and to adapt to partitioning changes at run time. We present such a barrier in the latter part of Section 3.4.3, enhanced with our optimal spin versus block decision-making technique within each processor.

3.3 Solution Structure

Because it is ultimately responsible for the fair allocation of resources among competing applications, a kernel-level scheduler cannot in general afford to accept arbitrary directives from user-level code. Our algorithms assume that processes can influence the behavior of the scheduler enough to avoid preemption in (short) critical sections. In general though the scheduler remains in control of processor allocation and the synchronization algorithms adapt to the current state of the machine. In order to drive that adaptation, scheduler-conscious synchronization algorithms require a mechanism to obtain information about the status of other processes and about the processors available to the application. This information may be (1) guessed via past experience using heuristics, (2) deduced through interaction with other processes, e.g. via "handshaking", or (3) provided by the kernel itself. In order, these options provide information of increasing accuracy and thus result in simpler algorithms and better performance.

Experience-based heuristics can be successful to the extent that the present and future resemble the past. They form the basis of both the competitive lock algorithms of Karlin et al. [KLMO91], and of the competitive barriers we describe at the beginning of Section 3.4.3. In the case of locks, the goal is to block if the wait time will be longer than twice the context switch time, and to spin if it will be shorter. For barriers, the goal is to block if there is another process (not currently running) that could use the current processor to make progress toward the barrier, and to spin otherwise. In both cases, the algorithm is able to determine (by reading the clock) whether blocking or spinning would have been a better policy at the most recent synchronization operation. If it finds it made the wrong decision, it biases its decision in favor of the other alternative the next time around. While this sort of adaptation has been shown to work better than any static alternative, it induces overhead to maintain the statistics that allow the decision to be made, and still makes the wrong decision some of the time.

Interaction with peer processes can provide better information about the peers' status, provided that they respond promptly to inquiries (when running). In Section 3.4.1 we use a "handshaking" technique in some of our mutual exclusion algorithms. To hand a peer a lock, a process sets a flag on which the peer is expected to be spinning, and then waits for the peer to set an acknowledgment flag. If the acknowledgment does not appear within a certain amount of time, the signaling process assumes that the peer is currently preempted. There is an inherent inefficiency with this approach: if the signaling process doesn't wait long enough, it will too often skip over a running peer by mistake. Any time it waits, however, is lost to computation. All the signaling process really needs to know is whether its peer is running or preempted, information readily available to the scheduler.

We have found heuristics and handshaking to be expensive both in implementation and execution cost. Algorithms based on kernel-provided information are simpler and easier to design. They generally provide superior performance, in part because the information from the kernel is more accurate than user-level estimates, and in part because the kernel can collect the information more efficiently than user-level code can guess it.

Our kernel extensions are enumerated below. They build upon ideas proposed by the Symunix project at NYU [ELS88]. Similar extensions could be based on the kernel interfaces of Psyche [MSLM91] or Scheduler Activations [ABLL92].[5]

- KE-1: For each process the kernel and user cooperate to maintain a variable that represents the process's state and that can be manipulated under certain rules by either. The variable has four possible values: `preemptable`, `preempted`, `self_unpreemptable`, and `other_unpreemptable`. `Preemptable` indicates that the process is running, but that the kernel is free to preempt it. `Preempted` indicates that the kernel has preempted the process. `Self_unpreemptable` and `other_unpreemptable` indicate that the process is running and should not be preempted. The kernel honors this request whenever possible (see KE-2 below), deducting any time it adds to the end of the current quantum from the beginning of the next.

  The `non-preemptable` states indicate that the process is executing in a critical section. Two distinct states are needed to accommodate certain race conditions in queue-based mutual exclusion algorithms, in which a process wishes to hand the lock to one of its peers and simultaneously make that peer unpreemptable. Most changes to the state variable make sense only for a particular previous value. For example, it makes no sense for user-level code to change a state variable from `preempted` to anything else. Overall system correctness does not depend on correct use of flags by applications, but the performance of a particular application may suffer if it uses the flags incorrectly. To make sure that changes happen only from appropriate previous values, our algorithms generally modify state variables using an atomic `compare_and_store` instruction.[6]

[5]If we were building on top of Psyche, we would replace code in our algorithms that sets the KE-1 variable to `self_unpreemptable` with code that blocks if the "two-minute warning" is in effect; we would delete code that resets KE-1 and blocks when KE-2 is set. Rather than change another process's KE-1 variable to `other_unpreemptable`, we would inspect its warning flag, and treat it as preempted if set. If we were building on top of Scheduler Activations, we would have the user-level scheduler that receives notice of kernel-initiated preemptions treat the KE-1 and KE-2 bits just as the kernel does in our Symunix-based proposal; it would need to resume execution of any "unpreemptable" process. Alternatively, we could have the user-level scheduler perform some synchronization algorithm-specific recovery, such as linking a process out of a queue, but this introduces additional complexity, because with Scheduler Activations the user-level scheduler itself can be preempted.

[6]`Compare_and_store(address,old_val,new_val)` compares `old_val` to the contents of

- KE-2: To ensure fairness for applications, the kernel maintains an additional per-process Boolean flag. This flag can be modified only by the kernel but is readable in user mode. The kernel sets a process's flag to indicate that it wanted to preempt the process but has honored a request (indicated via the KE-1 variable) not to do so. To maintain ultimate control of the processor, the kernel honors the request only when the KE-2 flag is not yet set; if the flag is already set the kernel proceeds with the preemption. Upon exiting a critical section (and setting the KE-1 variable back to `preemptable`), a process should inspect the flag and yield the processor if the flag is set. Yielding implicitly clears the flag. So long as critical sections are shorter than the interval between the kernel's attempts at preemption, voluntarily yielding the processor at the end of critical section in which the KE-2 flag has been set ensures that preemption will not occur during a subsequent critical section (barring page faults or other unusual sources of delay).[7]

- KE-3: The kernel also maintains a data structure, visible in user mode, that contains information about the hardware partition on which the application is running. Specifically, it contains the number of processors available in the partition, the `id` of the current processor, the number and ids of processes scheduled on each processor, and the *generation count* of the partition. The *generation count* indicates the number of times that the partition has changed in size since the application started running.

As noted above, extensions KE-1 and KE-2 are based in part on ideas developed for the Symunix kernel [ELS88]. We have introduced additional states, and have made the state variable writable and readable by both user-level and kernel-level code [WKS94]. Extension (KE-3) is a generalization of the interface described in our work on small-scale scheduler-conscious barriers [KW93] and resembles the "magic page" of information provided by the Psyche kernel [SLM90]. None of the extensions requires the kernel to do anything more often than once per quantum, or to maintain information that it does not already have available in its internal data structures. Furthermore, none requires the kernel or to access any user level code or data structures, or to understand the particular synchronization algorithm(s) being used by the application. We have run our experiments in user space, but a kernel-level implementation of our ideas would not be hard to build,

`address`. If they are identical it stores `new_val` in `address` and returns `true`. Otherwise it returns `false`.

[7]The possibility of page faults means that we cannot in general provide a guarantee against inopportune preemption. The best we can hope to do in any of our algorithms is to minimize the chance that such preemption will occur. To provide real guarantees (e.g. for a real time system), the kernel would need to ensure that a process that sets its KE-1 variable will always be able to execute some minimum number of instructions within a small bounded period of time.

```
type context_block = record
    state : (preempted, preemptable, unpreemptable_self, unpreemptable_other)
    warning : Boolean
    ...

type partition_block = record
    num_processors, generation : integer
    processes_on_processor : array [MAX_PROCESSORS] of integer
    processor_ids : array [MAX_PROCESSES] of integer
    ...
```

Figure 3.1: Pseudocode declarations for the kernel-application interface.

and could be expected to provide performance indistinguishable from that of our experimental environment.

Our scalable barrier and queue locks arrange for processors to spin only on local locations, on which no other processor spins. In most cases, we ensure that those locations will be local not only on cache coherent machines (on which they migrate to the spinning processor), but also on machines that lack hardware cache coherence. On these latter, NUMA machines, variables on which processes spin must be allocated statically in the local memory of the spinning processor; spins are terminated by a single uncached remote write by another processor.

3.4 Algorithms

In this section we present preemption-safe and scheduler-conscious synchronization algorithms that make use of heuristics, handshaking, and the extended kernel interface described in Section 3.3. We consider mutual exclusion, reader-writer locks, and barriers in turn. For the most part, we have chosen not to replicate the pseudocode here; it appears in a technical report [KWS94].

The pseudocode in Figure 3.1 defines the interface between the kernel and the application. The `state` field of a `context_block` is written by application processes to indicate when they do not want to be preempted. The remaining fields of both the `context_block` and `partition_block` records are writable only by the kernel scheduler; they provide the application with information about system state, to facilitate the design of efficient algorithms. Our mutual exclusion and reader-writer locks use only the `context_block` records; the barriers use both.

All of our algorithms work well in a dynamic hardware-partitioned environment, an environment widely believed to provide the best combination of throughput and fast turn-around for large-scale multiprocessors [CDD+91; LV90; TG89;

ZM90]. [8] Except for the barriers, which require partition information, all of the algorithms will also work well under ordinary time sharing. For a co-scheduled environment the additional complexity of preemption-safe and scheduler-conscious algorithms is not necessary, but does not introduce any serious overhead.

3.4.1 Mutual Exclusion

The ability to have single access to a shared resource is required by almost all parallel algorithms designed for SPARTAs. In applications ranging from parallel searching [SS89] to blackboard models [EL75; EHRLR80], the most frequent form of synchronization in a SPARTA is mutual exclusion. The parallel planner we describe in Chapter 2 makes use of mutual exclusion to protect a data structure of the best sheep save permutation. In real-time environments it is important to act in a scheduler-conscious manner to reduce the unpredictability otherwise encountered. To provide even greater predictability the queue locks we discuss in this chapter can be modified as suggested by Craig [Cra93] and Takada and Sakamura [TS94]. Providing scheduler-conscious mutual exclusion in SPARTAs is important. In this section we discuss how to achieve scheduler-conscious locks.

Scalability in mutual exclusion algorithms can be achieved by arranging for processes to spin on local locations in a distributed data structure, thereby eliminating interconnect and memory contention. Many researchers have developed algorithms of this type [And90; GT90; MLH94; MCS91a; YA93]. The scalability of centralized algorithms may also be improved by introducing appropriate forms of backoff [And90; MCS91a]. The preemption-safe `test_and_set` locks of Psyche, Symunix, or Scheduler Activations can be modified trivially to incorporate backoff, though the work of Anderson and of Mellor-Crummey and Scott suggests that the result will still produce more contention than a queue-based lock, degrading network response for other processes and applications.

In this section we present two scheduler-conscious variants of the queue-based lock of Mellor-Crummey and Scott. We also present a preemption-safe variant of the ticket lock. This latter lock, while less trivial than a preemption-safe `test_and_set` lock, is substantially simpler than a queue lock, and is likely to provide acceptable performance for many environments. Unlike a `test_and_set lock`, the ticket lock also provides fair FIFO ordering among currently-running processes. Because they awaken processes in a deterministic order, both the queue lock and the ticket lock must be modified to address not only preemption within a critical section, but also preemption while waiting in line.

Our first variant of the queue lock uses the Symunix kernel interface: kernel extension KE-2 and the `preemptable` and `self_unpreemptable` values (only) of

[8]Note that multiprogramming is still an issue on a partitioned machine, since an application with P processes may be forced to run in a partition with $< P$ processors.

KE-1. It uses handshaking to determine the status of other processes. When releasing a lock, a process notifies its successor in the queue that it (the successor) is now the holder of the lock. The successor must then acknowledge receipt of the lock by setting another flag. If this acknowledgement is not received within a fixed amount of time, the releasing process assumes that its successor is preempted, rescinds its notification, and proceeds to the following process (throughout this period the releasing process is unpreemptable).

Atomic `fetch_and_store` instructions are used to access the notification flag in order to avoid a timing window that might otherwise occur if the successor were to see its notification flag just before the releaser attempts to rescind it. Without the atomic instruction it would be possible for the releaser to think that the successor has failed and proceed to give the lock to another processor, and for the successor to think that it has succeeded and proceed to the critical section, thus violating mutual exclusion.

The handshaking version of the queue lock solves the preemption problem but unfortunately adds significant overhead to the common case. Processes need to interact several times when a lock is released. To address this limitation, we have designed a scheduler-conscious algorithm that uses the full version of kernel extension KE-1 and does not require handshaking. In this *Smart Queue* algorithm the releasing process examines its successor's state variable, which is kept up-to-date by the kernel. If the successor is preempted, the releaser proceeds to other candidates later in the queue. If the successor is running, the releaser uses an atomic `compare_and_store` instruction to change the successor's state to `other_unpreemptable`. If the change is successful the lock is passed to the successor. The need for `compare_and_store` stems from a potential race between the releaser and the kernel: after determining that the successor is not preempted, we must make it unpreemptable without giving the kernel an opportunity to preempt it.

One of the problems with queue-based locks is high overhead in the absence of contention. On small-scale machines and for low-contention locks a `test_and_set` with exponential backoff or ticket lock with proportional backoff may be preferable [MCS91a]. (A hybrid lock that switches between `test_and_set` and a queue-based lock, depending on observed contention, is another possibility [LA94].) With appropriate backoff, `test_and_set` and ticket locks scale equally well. They use different atomic instructions, making them usable on different machines. The ticket lock also guarantees FIFO service, while the `test_and_set` lock admits the possibility of starvation.

The basic idea of the ticket lock is reminiscent of the "please take a number" and "now serving" signs found at customer service counters. When a process wishes to acquire the lock it performs an atomic `fetch_and_increment` on a "next available number" variable. It then spins until a "now serving" variable matches the value returned by the atomic instruction. To avoid contention on large-scale machines, a process should wait between reads of the "now serving" variable for a period of time proportional to the difference between the last read value and the value returned by the `fetch_and_increment` of the "next available number" variable. To release the lock, a process increments the "now serving" variable.

Our preemption-safe, handshaking version of the ticket lock uses one additional "acknowledgment" variable, which contains the number of the last granted but un-acknowledged ticket. A releasing process sets the additional variable and the "now-serving" variable and waits for the former to be reset. If the acknowledgment does not occur within a timeout window, the releaser withdraws its grant of the lock, and re-increments the "now serving" variable in an attempt to find another acquirer. Changes to the acknowledgment variable are made with `compare_and_store` to avoid an update race between a skipped-over acquirer and its successor. Our ticket lock assumes that the "now serving" variable does not have the opportunity to wrap all the way around and reach a value it previously had, while a process remains preempted. For 32-bit integers, a 1 GHz processor, and an empty critical section, a process would have to be preempted for more than three minutes before correctness would be lost.

There is no obvious way to develop a preemption-safe or scheduler-conscious version of the ticket lock without either handshaking or exporting lock code into the kernel. The problem is that the lock does not keep track of the identities of waiting processes. The releaser of a lock is therefore unable to use KE-1 to determine the status of its successor: it does not know who the successor is.

A caveat with all three of our modified locks is that they give up the FIFO ordering of the scheduler-oblivious version. It is thus possible (though highly unlikely, and with probability lower than the probability of starvation with a test_and_set lock) that a series of adverse scheduling decisions could cause a process to starve. We have considered algorithms that leave preempted processes in an explicit queue so that they only lose their turn while they are preempted. Markatos adopted a similar approach in his real-time queue lock [ML91], where the emphasis was on passing access to the highest-priority waiting process. For simple unprioritized mutual exclusion, leaving preempted processes in the queue makes the common case more expensive: processes releasing a lock have to skip over their preempted peers repeatedly. We consider the (unlikely) possibility of starvation insignificant in comparison to this overhead.

The algorithms described in this section work only for single-nesting of locks. In the case of multiply-nested locks, a process should make itself preemptable only after releasing the outermost lock. This can be accomplished by keeping track of the nesting by incrementing a local variable on acquires and decrementing it on releases. The state flag should be set to preemptable only when the nesting reaches zero.

3.4.2 Reader-Writer Locks

Reader-writer locks are a refinement of mutual exclusion locks. They provide exclusive access to a shared data structure on the part of writers (processes making changes to the data), but allow concurrent access by any number of readers. There are several versions of reader-writer locks, distinguished by the policy they use to arbitrate among competing requests from both readers and writers. Reader-preference locks always force writers to wait until there are no readers interested in acquiring the lock. Writer-preference locks always force readers to wait until there are no interested writers. Fair variants prevent newly-arriving readers from joining an active reading session when there are writers waiting, and grant a just-released lock to the process(es) that have been waiting the longest.

Our scheduler-conscious reader-writer lock is based on a fair scalable reader-writer lock devised by Krieger et al. [KSU93]. When a writer releases a lock for which both readers and writers are waiting, and the longest waiting unpreempted process is a reader, the code grants access to all readers that have been waiting longer than any writers. An alternative interpretation of fairness would grant access in the same situation to *all* currently-waiting unpreempted readers. Like the Smart Queue lock, the reader-writer queue lock uses both kernel extensions KE-1 and KE-2.

Requests for the lock are inserted in a doubly linked list. A reader arriving at the lock checks the status of the previous request. If the previous request is an active reader or if there is no previous request, then the newly-arriving reader marks itself as an active reader and proceeds. In all other cases the newly-arriving process spins, waiting to be released by its predecessor. A process releasing a lock must first remove itself from the queue. If the process is a writer this is an easy task since it has no predecessor in the queue and the procedure is similar to the one followed in the mutual exclusion section. If it is a reader however, then the process may have to remove itself from the middle of the queue. To ensure correct manipulation of the linked list data structure a reader process locks both its own list node and that of its predecessor. It then updates the link pointers to reflect the new state of the list. The locks protecting individual list elements use `test_and_set`. We have opted for this type of lock because the critical sections are short and the maximum number of contending processes is three. When an unlocking reader attempts, unsuccessfully, to acquire the lock on its predecessor's list element, it re-checks the identity of the predecessor in case it has changed as a result of action on the part of whoever was holding the lock.

After a process has linked itself out of the queue, it must wake up its successor if there is one. The procedure is similar to the one followed in the mutual exclusion case. The releasing process checks the state of its successor and attempts to set its state to `unpreemptable_other`. If the attempt is successful, the releasing process proceeds to notify its successor that it has been granted the lock. If the attempt fails, it notifies its successor of failure by setting a flag in the successor's node, and proceeds to the next process in the queue. When notified that it has been granted the lock, a reader uses this same procedure to release its own successor, if that successor is also a reader.

3.4.3 Barriers

We discuss three types of preemption-safe and scheduler-conscious barriers in this section. The first two types are designed for bus-based multiprocessors, or for small partitions on larger machines, in which migration is assumed to be relatively inexpensive. They differ in the amount of information they use in order to make their decisions, and in the quality of those decisions. The first type requires no kernel extensions; it employs heuristics and comes in several variants. The second type employs kernel extension KE-3 to make optimal spin versus block decisions. The third type of barrier is designed for large-scale multiprocessors, on which migration is assumed to be an expensive, uncommon event. This barrier makes optimal spin versus block decisions within each processor (or within each cluster of a machine in which migration is inexpensive among small sets of processors), uses a logarithmic-time scalable barrier across processors/clusters, and adapts dynamically to changes in the allocation of processes to processors or processors to applications.

Barrier synchronization's primary source of performance loss in a multiprogrammed environment is the cycles wasted spinning while waiting for preempted processes to arrive at the barrier. In order to reduce the performance penalty of wasted spinning, processes can choose to block and relinquish their processor to a preempted peer. Blocking however can be expensive, especially on modern processors, due to the large amount of state that needs to be saved. There are several possible ways to resolve this tradeoff, ranging from always spin to always block. For an environment that is not known at compile time, neither extreme provides a satisfactory solution.

Inspired by the work of Karlin et al. for locks [KLMO91], we have proposed several variants of a competitive centralized barrier [KW93]. The simplest variant spins for a fixed amount of time and then blocks. By setting the spin time equal to the context switch time, we can guarantee that the algorithm takes at most twice as long as necessary. More complicated variants gather information from a small number of recent barrier episodes and shorten or lengthen their spinning threshold based on the observed waiting time. In effect, they use the waiting time at recent barriers to guess whether the machine is multiprogrammed. If waits have been long, processes guess that they are sharing the processor, and should therefore block at a barrier; otherwise they spin.

The competitive barriers require no kernel extensions, and often work quite well. They suffer from several limitations, however. The variant that always spins for the duration of a context switch always wastes that time in a multiprogrammed environment. The experience-based variants, on the other hand, make correct decisions only if the number of available processors changes infrequently, compared to the rate at which barriers are encountered. They also count on the work between barrier episodes being more-or-less equal across processes: variations in workload result in arrival time skews that can cause them to guess that multiprogramming is happening when it is not. In this case, their performance degrades to that of "always block."

Finally, all of the competitive barriers use a uniform policy for all processes: either all will spin or all will block. Ideally on a system with N processes and P processors (and inexpensive migration) the first $N - P$ processes should block while the remaining P should spin. By using kernel extension KE-3, we can keep the application appraised of the number of available processors. Then, since centralized barriers already keep a count of the number of processes that have arrived at the barrier so far, and since they can easily incorporate knowledge of the total number of processes, each process arriving at the barrier can make the optimal choice as whether to spin or to block. This *Scheduler Information* barrier has low overhead (a check against the number of available processors) and makes the optimal spin versus block decision, and it makes this decision on a process by process basis.

Barriers for large-scale multiprocessors are of necessity more complicated since counter-based algorithms are too slow (linear in the number of processes) and cause too much contention due to the bottleneck of the counter. Scalable barrier algorithms use log-depth data structures to register the arrival and signal the departure of processes. Unfortunately, these data structures tend to exacerbate the problems caused by multiprogrammed environments, since they require portions of the barrier code in different processes to be interleaved in a deterministic order. This order may conflict with the scheduling policy on a multiprogrammed system, causing an unreasonable number of context switches [ML91] to occur before achieving the barrier.

The basic idea of our scalable scheduler-conscious barrier is to make the optimal spin versus block decision within each individual processor or cluster, and to employ a scalable log-depth barrier across processors, where context switches are not an issue. Specifically, we combine the Scheduler Information barrier described above with the scalable tree barrier of Mellor-Crummey and Scott [MCS91a]. Processes assigned to the same processor or cluster use a Scheduler Information barrier. The last process to reach the barrier becomes the representative process for the processor/cluster. Representative processes participate in the tree barrier.

A partition generation count allows us to handle repartitioning—changes in the mapping of processes to processors or clusters. We shadow this generation count with a count that belongs to the barrier. The process at the root of the inter-processor barrier checks the barrier generation count against the partition generation count. If the two counts are found to be different, processes within each new processor/cluster elect a representative and the representatives then go through a barrier reorganization phase, initializing tree pointers appropriately.[9] This approach has the property that barrier data structures can be reorganized only at a barrier departure point. As a result, processes may go through one episode of the barrier using outdated information. While this does not affect correctness it could have an impact on performance. If repartitioning were a frequent event, then processes would use old information too often and performance would suffer. However, we consider it unlikely that repartitioning would occur more than a few times per second on a large-scale, high-performance machine, in which case the impact of using out-of-date barrier data structures would be negligible.

3.5 Results

This section presents a performance evaluation of different preemption-safe and scheduler-conscious synchronization algorithms, including a comparison to the best known scheduler-oblivious algorithms. We begin by describing our experimental methodology. We then consider mutual exclusion, reader-writer locks, and barriers in turn.

3.5.1 Methodology

We have tested our algorithms on two different architectures. As an example of a small-scale bus-based machine we use a 100 MHz, R4400 12-processor Silicon Graphics Challenge. As an example of a large-scale distributed memory machine we use a 64-processor partition of a 20 MHz Kendall Square KSR 1. We have used both synthetic and real applications. The synthetic applications allow us to thoroughly explore the parameters that may affect synchronization performance, including the ratio between the lengths of critical and non-critical sections, the degree of multiprogramming, the quantum size, and others. The real applications allow us to validate our findings in the context of a larger computation, potentially capturing effects that are missing in the synthetic applications, and providing a measure of the impact of the synchronization algorithms on overall system performance. We have chosen applications that make heavy use of synchronization constructs to ensure that synchronization time is a significant portion of program

[9]Note that the representative for the re-organization phase is not necessarily the process that will participate in the inter-processor phase of subsequent barriers; this latter role is played by the last process to arrive at each individual intra-processor barrier.

runtime. For applications that make little use of synchronization constructs, we expect that the choice of the synchronization algorithm will have little or no impact on performance.

Our synchronization algorithms employ atomic operations not available on either of the two target architectures. We have implemented software versions of these instructions using `load_linked` and `store_conditional` on the Challenge and small critical sections bracketed by the native synchronization primitive (`get_subpage` and `free_subpage`) on the KSR. Our approach for the KSR adds overhead to the algorithms, but this overhead is small. Moreover, because we are running scalable algorithms, in which processes use backoff or spin only on local locations, competition is essentially non-existent for the critical sections that implement the "atomic" operations, and does not result in any significant increase in overall levels of network and memory contention. Our results for the non-native locks are therefore slightly higher in absolute time, but qualitatively very close in character, to what would be achieved with hardware supported `fetch_and_Φ` instructions.

For the sake of simplicity, we employ a user-level scheduler in our experiments. One processor is dedicated to running the scheduler. While the kernel interface described in Section 3.3 would not be hard to implement, it was not needed for our experiments, and we lacked the authorization to make kernel changes on the KSR. For the lock experiments, each application process has its own processor. Preemption is simulated by sending a signal to the process. The handler for this signal spins on a flag that the scheduler process sets when it is time to return to executing application code. The time spent spinning is meant to represent execution by one or more processes belonging to other, unrelated applications.

For the scalable barriers, multiprogramming is simulated by multiplexing one or more application processes on the same processor. Both the SGI and KSR operating systems allow us to do this by binding processes to processors. The centralized barrier experiments require process migration. On the SGI we can restrict processors (prevent processes from executing on them). Restricting a processor increases the multiprogramming level on the remaining processors. Processes are allowed to migrate among the unrestricted processors. The KSR operating system does not provide an analogue of the SGI restrict operation, so we were unable to control the number of processors available to migrating processes. For this reason we do not report results for the centralized barriers on the KSR.

The *multiprogramming level* reported in the experiments indicates the average number of processes per processor. For the lock-based experiments, one of these processes belongs to the application program; the others are assumed to belong to other applications, and are simulated by spinning in a signal handler as described above. For the barrier-based experiments, multiple application processes reside on each processor, and participate in all the barriers. The reason for the difference in methodology is that for lock-based applications we are principally concerned about processes being preempted while holding a critical resource, while for barrier-based applications we are principally concerned about processes wasting processor resources while their peers could be doing useful work. Our lock algorithms and the small-scale barriers are designed to work in any multiprogrammed environment; the scalable barrier assumes that processors are partitioned among applications. A multiprogramming level of 1 indicates one application process on each processor. Fractional multiprogramming levels indicate additional processes on some, but not all, processors.

In most respects we believe that performance results on a real implementation of our kernel extensions would be indistinguishable from those reported here; the scheduler itself does very little work, and only once per quantum. The one exception arises in the lock experiments, where simulation of preemption via spinning in a signal handler fails to capture any delays due to loss of cache, TLB, or memory footprint during preemption. Since these effects are inherently dependent on the memory reference characteristics of whatever unrelated processes happen to be running on the machine, they would be difficult to model in any experimental setting.

3.5.2 Mutual Exclusion

We implemented ten different mutual exclusion algorithms:

TAS-B – A standard test-and-`test_and_set` lock with bounded exponential backoff. This algorithm repeatedly reads a central flag until it appears to be unset, then attempts to set it atomically in order to acquire the lock. On the SGI Challenge, this is the native lock, augmented with backoff.

TAS-B-np – The same as TAS-B, but avoids preemption in critical sections by using the Symunix kernel interface.

Queue – A list-based queue lock with local-only spinning [MCS91a].

Queue-np – An extension to the Queue lock that avoids preemption in critical sections, also using the Symunix kernel interface. This algorithm does *not* avoid passing the lock to a process that has been preempted while waiting in line.

Queue-HS – An extension to the Queue-np lock that uses handshaking to ensure that the lock is not transferred to a preempted process. This is the first algorithm described in Section 3.4.1.

Smart-Q – An alternative extension to the Queue-np lock that uses kernel extensions KE-1 and KE-2 to obtain simpler code and lower overhead than in the Queue-HS lock. This is the second algorithm described in Section 3.4.1.

Ticket – The standard ticket lock with proportional backoff, but with no special handling of preemption in the critical section or the queue.

Ticket-np – A preemption-safe ticket lock with backoff, using handshaking to avoid preemption in the critical section. This is the third algorithm described in Section 3.4.1.

Native – A lock employing machine-specific hardware. This is the standard lock that would be used by a programmer familiar with the machine's capabilities. It does not incorporate backoff.

Native-np – An extension to the native lock that uses the Symunix kernel interface to avoid preemption while in the critical section.

The Native lock on the SGI Challenge is a test-and-`test_and_set` lock implemented using the `load_linked` and `store_conditional` instructions of the R4400 microprocessor. The Native lock on the KSR 1 employs a cache line locking mechanism that provides the equivalent of queue locks in hardware. The queuing is based on physical proximity in a ring-based interconnection network, rather than on the chronological order of requests.[10] We would expect the Native-np locks to outperform all other options on these two machines, not only because they make use of special hardware, but because the atomic operations in all the other locks are built on top of them. Our experiments confirm this expectation.

Our synthetic application executes a simple loop consisting of a work section and a critical section. The total number of loop iterations is proportional to the number of executing processes. In designing a scalable synthetic application we needed to ensure that the critical section did not become a bottleneck. Therefore, we set the ratio of the lengths of the critical and non-critical sections on both machines to slightly less than the inverse of the maximum number of processors. Absolute quantum length (in cycles or microseconds) had no significant effect on performance. We therefore concentrate here on the remaining variables in the synthetic application: multiprogramming level and number of processors.

Figures 3.2 and 3.3 plot execution time of the synthetic application against multiprogramming level for a fixed number of processors (11 on the SGI and 63 on the KSR). On the SGI, the scheduling quantum is fixed at 20 ms and the critical to non-critical section ratio is 1:14. We used a random number generator to vary the length of the critical section within a narrow range, to more closely

[10]We use the KSR's **gspnwt** instruction in a loop, rather than **gspwt**. Counterintuitively, the latter does not perform well when there are more than a handful of contending processors.

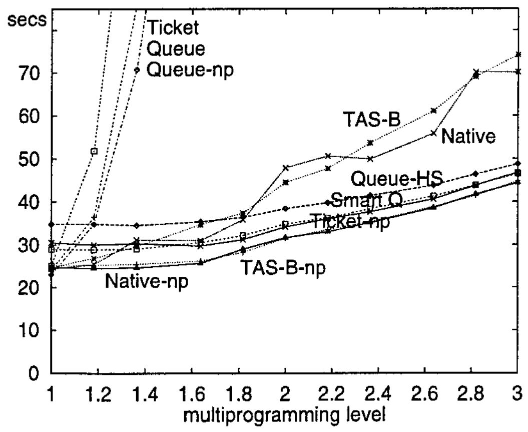

Figure 3.2: Varying multiprogramming level on an 11-processor SGI Challenge.

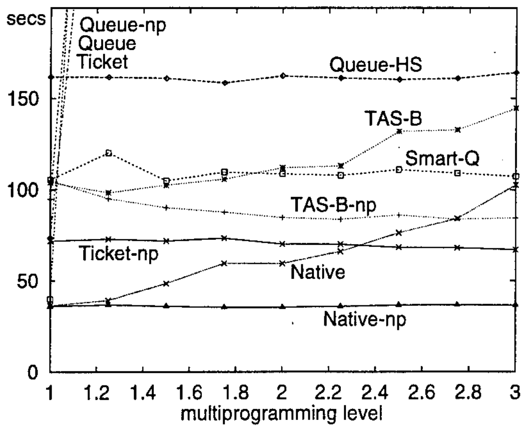

Figure 3.3: Varying multiprogramming level on a 63-processor KSR 1.

approximate the behavior of real applications and to avoid possible lock-stepping of the different processes. The Queue, Queue-np, and Ticket locks show the worst degradation, because processes queue up behind preempted peers. Preventing preemption in the critical section helps a little, but not much: preemption of processes waiting in the queue is the dominant problem.

Considerably better behavior is obtained by preventing critical section preemption *and* ensuring that the lock is not given to a blocked process waiting in the queue: the Queue-HS, Smart-Q, and Ticket-np locks perform far better than the other scalable locks, and also outperform the Native and TAS-B locks at multiprogramming levels greater than 2. The Native-np and TAS-B-np locks display the best results, though past work [MCS91a] suggests that they will generate

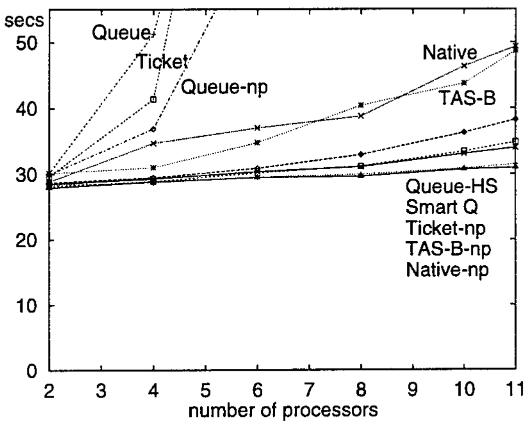

Figure 3.4: Varying the number of processors on the SGI Challenge with a multiprogramming level of 2.

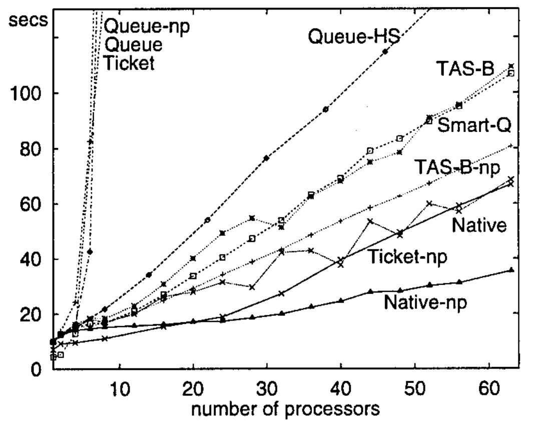

Figure 3.5: Varying the number of processors on the KSR 1 with a multiprogramming level of 2.

more bus traffic than the scalable locks, and would interfere more with ordinary memory accesses in other processes or applications.[11]

On the KSR, the scheduling quantum is fixed at 50 ms and the ratio of critical to non-critical section lengths is 1:65. The results show slightly different behavior from that on the SGI. The Queue, Queue-np, and Ticket locks suffer an even greater performance loss as the multiprogramming level increases. The Queue-HS lock improves performance considerably, since it eliminates both the critical section and queue preemption problems. Unfortunately, it requires a significant number of high-latency remote references, resulting in a high, steady level of overhead. The Smart-Q lock lowers this level by a third, but it is still a little slower than the TAS-B-np lock. The best non-native lock is Ticket-np.

The Native-np lock provides the best overall performance. Since all the non-native locks use native locks internally to implement atomic operations, this is expected behavior. The TAS-B and Native locks perform well when the multiprogramming level is low, but deteriorate as it increases. If the necessary atomic operations (`fetch_and_add`, `swap`, etc.) were available on the KSR 1, we would expect the queue and ticket locks to perform better than they do by a small constant factor. The closeness with which those locks follow the performance of KSR's relatively complex built-in primitive suggests that that primitive is probably not cost effective.

Increasing the number of processors working in parallel can result in a significant amount of contention, especially if the program needs to synchronize frequently. Previous work has shown that queue locks improve performance in such an environment, but as indicated by the graphs in Figures 3.2 and 3.3 they experience difficulties under multiprogramming. The graphs in Figures 3.4 and 3.5 show the effect of increasing the number of processors on the different locks at a multiprogramming level of 2.

The synthetic program runs a total number of loop iterations proportional to the number of processors, so execution time should not decrease as processors are added. Ideally, it would remain constant, but contention and scheduler interference will cause it to increase. With quantum size and critical to non-critical ratio fixed as before, results on the SGI again show the Queue, Queue-np, and Ticket locks performing poorly, as a result of untimely preemption. The performance of the TAS-B and Native locks also degrades with additional processors, because of increased contention and because of the increased likelihood of preemption in the critical section. The Smart-Q and Ticket-np locks degrade more slowly, but also appear to experience higher overheads. Increasing the number of processors

[11]The synthetic application does not capture this effect; it operates almost entirely out of registers during its critical and non-critical sections. The impact on data-access traffic can be seen in our experiments with real applications.

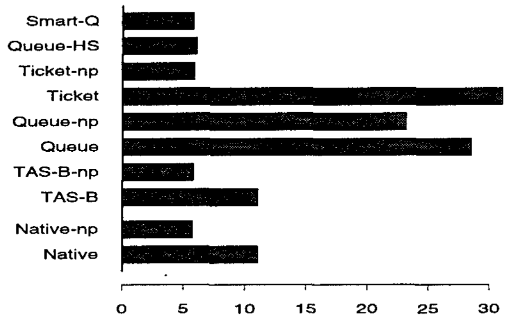

Figure 3.6: Completion time (secs) for Cholesky on the SGI using 11 processors (multiprogramming level = 2).

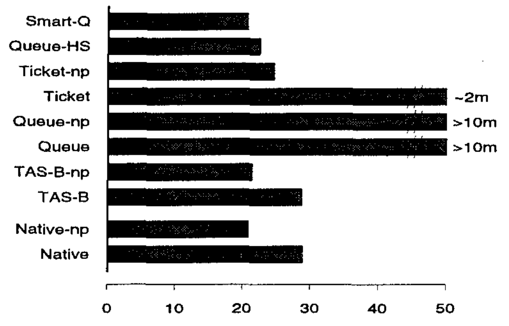

Figure 3.7: Completion time (secs) for Cholesky on the KSR using 63 processors (multiprogramming level = 2).

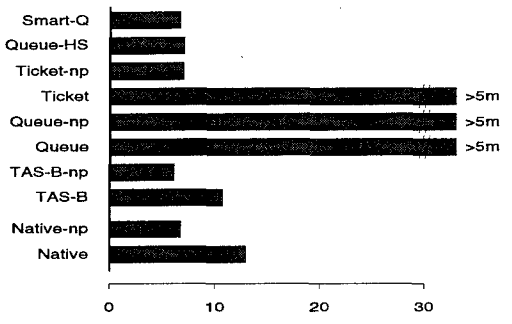

Figure 3.8: Completion time (secs) for Quicksort on the SGI using 11 processors (multiprogramming level = 2).

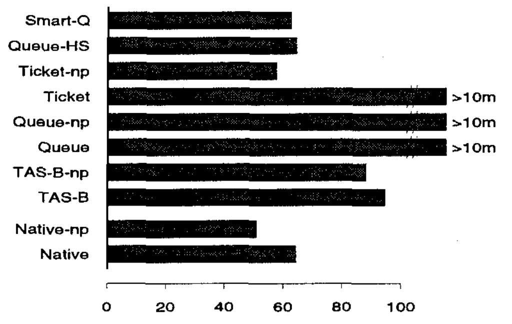

Figure 3.9: Completion time (secs) for Quicksort on the KSR using 63 processors (multiprogramming level = 2).

does not affect the TAS-B-np and Native-np locks until there are more than about eight processors active (the point at which bus contention becomes an issue).

The results on the KSR indicate that contention effects are important for larger numbers of processors. The native lock, with our modification to avoid critical section preemption, is roughly twice as fast as the nearest competition, because of the hardware queuing effect. Among the all-software locks, Ticket-np performs the best but TAS-B-np and Smart-Q are still reasonably close.

Backoff constants for the TAS-B and Ticket locks were determined by trial and error. The best values differ from machine to machine, and even from program to program. The queue locks are more portable. As noted above, contention on both machines becomes a serious problem sooner if the code in the critical and non-critical sections generates memory traffic. Results from real applications indicate that the queue locks suffer less from this effect.

To verify the results obtained from the synthetic program, and to investigate the effect of memory traffic generated by data accesses, we measured the performance of three real applications: the Cholesky program from the Stanford SPLASH suite [SWG92] running on matrix `bccstk15`, a multiprocessor version of Quicksort on 2 million integers, and a program that solves the traveling salesperson (TSP) problem for a 17-city fully connected graph. These programs contain no barriers; they synchronize only with locks. The TSP and Quicksort programs have similar performance. Figures 3.6 to 3.9 show the completion times for the remaining two applications, in seconds, when run with a multiprogramming level of 2 using 11 processors on the SGI and 63 processors on the KSR. As with the synthetic program, multiprogramming was simulated by spinning in a signal handler when other applications were supposed to be running. Again we see that scheduler-oblivious queuing of preemptable processes is disastrous. This time, however, with real computation going on, the Ticket-np and Smart-Q locks match the performance of the TAS-B-np and Native-np locks on the SGI, and outperform TAS-B-np in Quicksort on the KSR. We also ran experiments with a multiprogramming level of 1. Results (not shown) indicate that Quicksort and TSP run about 10% slower when using the Queue-HS lock than they do with the regular Queue lock. Otherwise, performance differences between applications with preemption-safe or scheduler-conscious locks and applications with the corresponding scheduler-oblivious locks were negligible.[12]

We have also collected single-process latency numbers (i.e. the time to acquire and release a lock in the absence of competition for the lock) to establish the performance overhead of the preemption-safe and scheduler-conscious algorithms with respect to their scheduler-oblivious counterparts.[13] Results appear in table 3.10. They were collected by having a single process acquire the lock repeatedly in a loop. As a result, they do not count the time required to bring the lock into the cache if it was most recently accessed by a different processor.

In their original, scheduler-oblivious form, queue locks have roughly an additional 8% overhead over centralized (ticket or test-and-`test_and_set`) locks. Adding code to avoid inopportune preemption adds no more than 9% to the cost of the queue locks, but significantly raises the cost of the centralized locks, not only because they are faster to begin with, but also because they must pay the overhead of scheduler consciousness in all cases, while the queue locks are able to skip most of the special-purpose code when the queue of waiting processes is

[12]In addition to a lock-protected work queue, TSP uses 5 atomic counters, which we implemented with `fetch_and_add`. Implementing them with critical sections instead, dramatically increases the impact of synchronization on program run-time. In this case, TSP runs an additional 10% slower when using the Queue-HS or Smart-Q locks than it does with the regular Queue lock.

[13]At the time we collected the latency numbers, the KSR had been decommissioned, so we were able to collect them only on the SGI. However since these are single-processor experiments we do not believe that the KSR numbers would have added anything significant to the understanding of the algorithms.

Lock	Latency (μs)	
TAS-B	2.10	
TAS-B-np	2.47	.37 = 18%
Ticket	2.10	
Ticket-np	2.87	.77 = 36%
Queue	2.26	
Queue-HS	2.46	.20 = 9%
Smart-Q	2.44	.18 = 8%
native	2.04	
native-np	2.39	.35 = 17%

Figure 3.10: Latency (acquire + release) of mutual exclusion locks on the SGI. The extra numbers in the right-hand column indicate the absolute and percentage increase in latency of the preemption-safe and scheduler-conscious locks with respect to their scheduler-oblivious counterparts.

empty.

3.5.3 Reader-Writer Locks

We implemented six different reader-writer locks:

Lock	R-Lat (μs)		W-Lat (μs)	
RW-TAS-B	3.13		2.10	
RW-TAS-B-np	3.15	.02 = 0%	2.52	.40 = 19%
RW-Queue	5.28		2.21	
RW-Smart-Q	5.48	.20 = 4%	2.67	.46 = 21%

Figure 3.11: Latency (acquire + release) of reader-writer locks on the SGI. The extra numbers in the second and third columns indicate the absolute and percentage increase in latency of the preemption-safe and scheduler-conscious locks with respect to their scheduler-oblivious counterparts, for both the reader and writer parts of the algorithms.

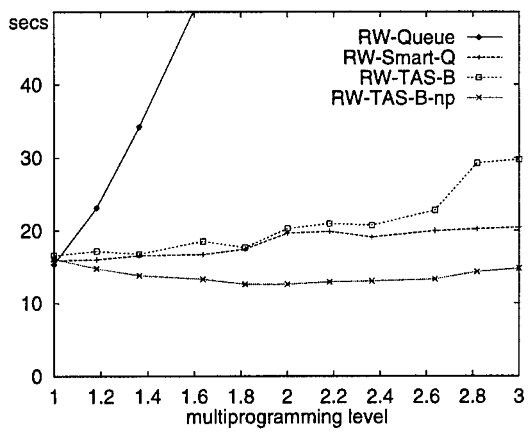

Figure 3.12: Varying the multiprogramming level for the reader-writer lock on the SGI (11 processors).

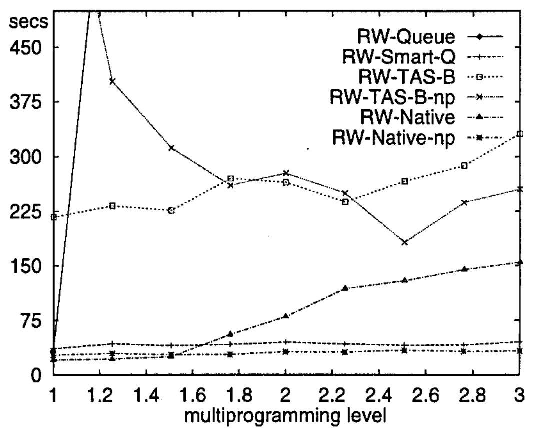

Figure 3.13: Varying the multiprogramming level for the reader-writer lock on the KSR (63 processors).

RW-TAS-B – A centralized reader-writer lock based on a standard test-and-test_and_set lock with exponential backoff.

RW-TAS-B-np – The same as RW-TAS-B, but with avoidance of preemption in critical sections, using the Symunix kernel interface.

RW-Queue – A scalable reader-writer lock based on the lock by Krieger et al. [KSU93].

RW-Smart-Q – An extension to the RW-Queue lock that uses kernel extensions KE-1 and KE-2 to avoid preemption in the critical section, and to avoid

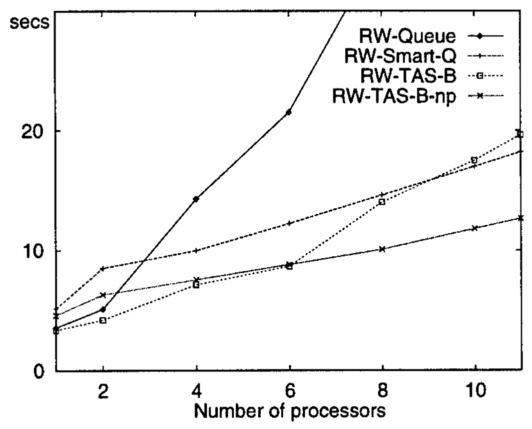

Figure 3.14: Varying the number of processors for the reader-writer lock on the SGI (multiprogramming level = 2).

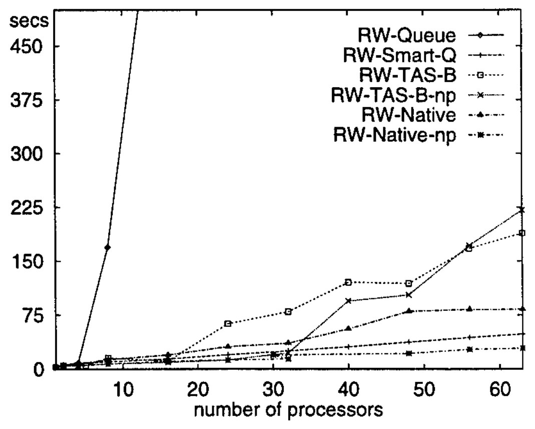

Figure 3.15: Varying the number of processors for the reader-writer lock on the KSR (multiprogramming level=2).

passing the lock to a preempted process. This is the algorithm described in Section 3.4.2.

RW-Native – A reader-writer lock based on the native synchronization primitive. On the SGI this is identical to the RW-TAS-B lock.

RW-Native-np – The same as RW-Native, but with avoidance of preemption in critical sections, using the Symunix kernel interface. On the SGI this is identical to the RW-TAS-B-np lock.

Figures 3.12 and 3.13 show the performance of the various reader-writer locks under varying levels of multiprogramming on the SGI (11 processors) and KSR (63 processors), respectively. Figures 3.14 and 3.15 show performance on varying numbers of processors, at a multiprogramming level of 2.

Reader-writer locks display behavior similar to that of mutual exclusion locks. The RW-Native-np lock outperforms all the others in a multiprogrammed environment. The RW-Smart-Q lock is a close second. The algorithms that do not cope with preemption behave increasingly worse as the multiprogramming level increases, though this effect is less pronounced than it was in the case of mutual exclusion. Five percent of the critical sections in our experiments acquire a writer lock; the rest acquire a reader lock, and can proceed in parallel with other readers. Preempting a process that holds a lock usually means preempting a reader, not a writer, so other readers can still proceed (so long as a writer is not yet in line).

For completeness, we ran experiments with one, five, and fifty percent writers. Larger numbers of writers cause a higher degree of contention—expected since there is less concurrency available—and degrade the performance of the RW-TAS-B and RW-TAS-B-np locks. We present the five percent results here. The others are qualitatively similar.

As in the case of mutual exclusion, the centralized RW-TAS-B and RW-TAS-B-np locks still suffer from contention on large numbers of processors. Contention effects are more pronounced than they were for mutual exclusion. With the ratio of critical to non-critical work the same as in the mutual exclusion experiments, we expected that the additional parallelism available due to the concurrency of readers would reduce the observed contention, but this turned out not to be the case. Graph 3.13 shows that the centralized RW-TAS-B-np lock actually improves in performance as multiprogramming increases. With fewer processes running in parallel, reductions in contention allow lock operations to complete faster, even though there are fewer total cycles available to the application per unit of time.

We have also collected single-process latency numbers for both the reader and writer parts of the locks. Results appear in table 3.11. They were collected by having a single process acquire and release the lock repeatedly in a loop. As a result, they do not count the time required to bring the lock into the cache if it was most recently accessed by a different processor. Most of the additional complexity of the preemption-safe and scheduler-conscious versions of the reader locks appears in the code for writers, rather than readers. Moreover, since the scheduler-oblivious overhead for readers is already nearly 50% higher than the cost of a mutual exclusion lock, the percentage increase in latency for readers when moving to a preemption-safe or scheduler-conscious lock is insignificant. The increase in latency for writers is on the order of 20%.

3.5.4 Barriers

We present results on barrier synchronization in two sections, one for small-scale machines such as the SGI Challenge (these results also apply to small partitions of a larger machine), and one for large-scale machines such as the KSR 1.

Small-Scale Barriers

For small-scale, centralized barriers, we implemented three baseline cases—always spin, always block, and spin-then-block—, three competitive algorithms that adjust their behavior based on previous barrier episodes, and the Scheduler Information barrier of Section 3.4.3, which uses kernel extension KE-3 to make an optimal spin versus block decision:

C-spin – All processes spin while waiting for their peers to reach the barrier.

C-block – Processes never spin; if they need to wait, they place themselves on a semaphore queue. The last process to arrive at the barrier wakes up its peers by performing V operations on the semaphore.

C-sp-blk – Processes spin for a bounded amount of time equal to the cost of a context switch. If the bound expires before the barrier is achieved, then the process yields the processor by performing a P operation on a semaphore. The last process to arrive at the barrier checks the semaphore queue and wakes up any processes that are blocked.

C-last1 – Processes spin for a period of time determined by how long they waited at the previous barrier episode. This time is increased if the process did not exceed it during the last barrier episode and decreased otherwise. The upper bound on the time is the cost of a context switch.

C-avg3 – This barrier is similar to the C-last1 barrier, except that the last three barrier episodes are used in determining the amount of time spent spinning.

C-coarse – A competitive barrier similar to the C-last1 barrier, except that the spinning time is not adjusted incrementally, but rather all at once.

C-sched – A barrier that makes an optimal spin versus block decision based on the number of available processors (as reported by kernel extension KE-3) and the number of processes that have yet to reach the barrier.

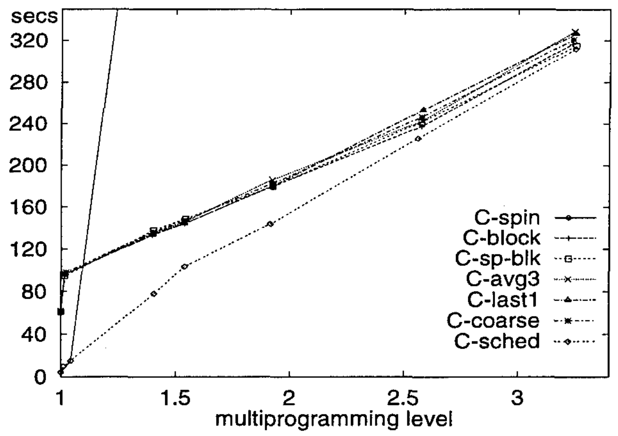

Figure 3.16: Performance of the small-scale barriers for the synthetic program.

Figure 3.16 shows the performance of the synthetic application on the SGI Challenge when using different barrier implementations, as the multiprogramming level increases. Our experiments were run in a dynamic hardware partitioned environment, where the number of processors available to the application varies between 5 and 11, with an average of 7.9. The multiprogramming level is calculated based on the average number of processors and the number of processes used by the application. The synthetic application performs no real work between barriers; it does not capture the effect of data-access memory references. The kernel scheduler moves processes among processors in order to balance load. Processes running on the same processor are multiprogrammed with a quantum length of 30 ms, the default value used by the IRIX kernel scheduler.

In the absence of multiprogramming, the C-sched barrier performs as well as the C-spin barrier—its overhead is low—, and significantly better than the competitive barriers. As the multiprogramming level increases the spinning barrier's performance degrades sharply, while the C-sched barrier retains its good performance and its advantage over the other algorithms. It never spins when other processes could make use of the current processor, and it avoids the overhead of blocking in the last P processes to arrive at the barrier. At very high multiprogramming levels, the Scheduler Information barrier is only slightly faster than the competitive and blocking barriers: as the number of processes per processor increases it becomes less important to avoid blocking in the final process on each processor.

To validate the results obtained with the synthetic application, we experimented with two real applications as well: Gaussian elimination and Successive

Over-Relaxation (SOR). Gauss solves a 640×640 problem without pivoting; SOR works on a 800×800 matrix for 20 iterations. Both applications use 11 processes on 5–11 processors, with a quantum length of 30 ms and a repartition operation (a random change in the number of processors) every 80 ms.

The main difference we observed with respect to the synthetic results is a decrease in the impact of synchronization on overall performance, since it is combined with the time spent in real computation. Figures 3.17 and 3.18 show the completion time of Gaussian elimination at multiprogramming levels of 1 and 2 respectively; the corresponding results for SOR appear in Figures 3.19 and 3.20. In both cases the C-sched barrier provides the best performance. In SOR, however, the heuristic barriers also provide good performance in both the multiprogrammed and dedicated environments, while in Gauss they track the performance of the blocking barrier. The reason is that in Gauss processes arrive at the barrier skewed in time, due to the different amount of work they have to do. The heuristic barriers wrongly assume that the skew is due to multiprogramming and resort to blocking.

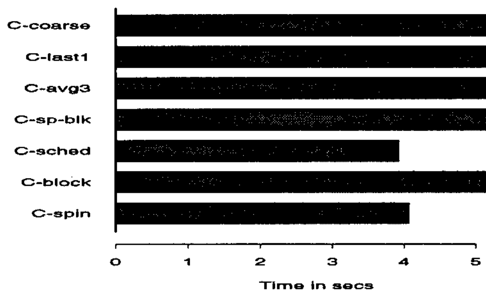

Figure 3.17: Gaussian Elimination runtime for different barrier implementations (multiprogramming level=1).

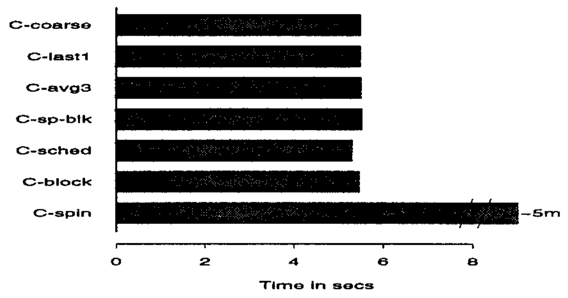

Figure 3.18: Gaussian Elimination runtime for different barrier implementations (multiprogramming level=2).

As with mutual exclusion and reader-writer locks, we experimented with a variety of other values for each of the experimental parameters. The only parameter (other than multiprogramming level and number of processors) to display a noticeable impact on performance was the frequency of repartitioning decisions. As the time between repartitions increases, the performance of the competitive barriers improves to some extent, since they need time to adapt to a change in partition size, and an increase in the time between repartitions allows them to amortize their adaptation cost over a larger number of episodes. The performance of the blocking and spinning barriers is essentially independent of the time between repartitions.

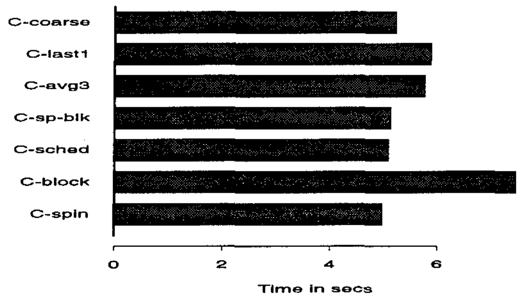

Figure 3.19: SOR run-time for different barrier implementations (multiprogramming level=1).

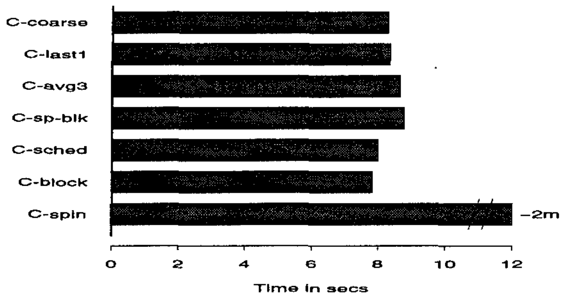

Figure 3.20: SOR run-time for different barrier implementations (multiprogramming level=2).

We were initially surprised to see a small but steady improvement in the performance of the C-sched barrier as the time between repartitions increased. The explanation is that it is possible for the algorithm to err when a repartitioning decision occurs at the same time that the application is going through a barrier. In this case, some threads will use old information to guide their decision and thus may decide sub-optimally. When the time between scheduling decisions is large, sub-optimal decisions happen less frequently, resulting in a small performance improvement.

Scalable Barriers

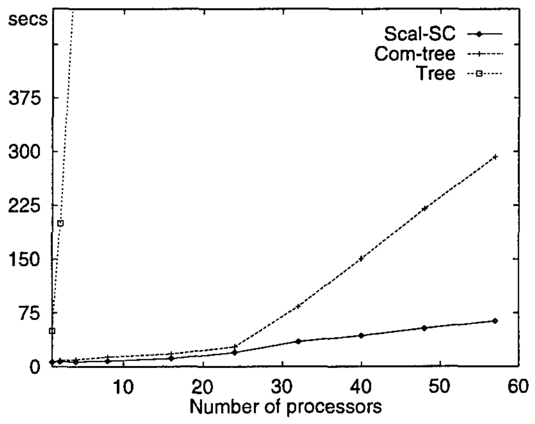

Figure 3.21: Varying the number of processors for the barriers on the KSR (multiprogramming level = 2).

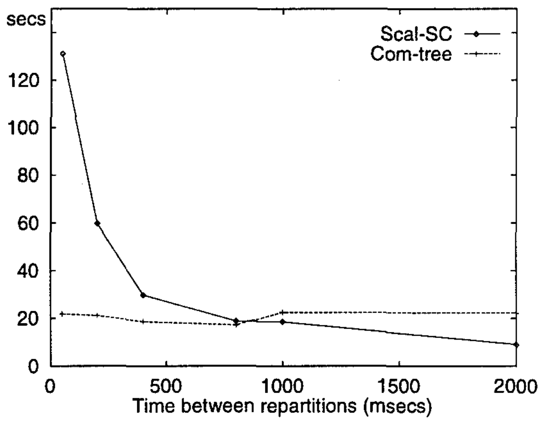

Figure 3.22: Varying the frequency of repartitioning decisions for the barriers on the KSR (57 processors).

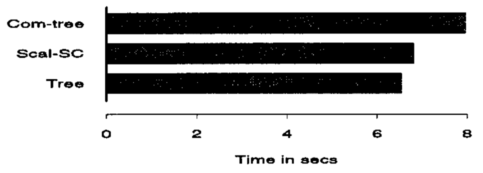

Figure 3.23: Gaussian elimination runtime on the KSR using 57 processors (multiprogramming level = 1)

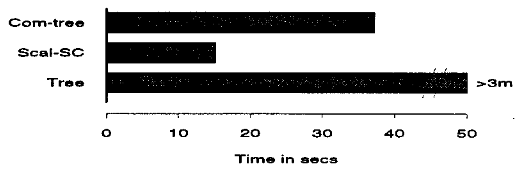

Figure 3.24: Gaussian elimination runtime on the KSR using 57 processors (multiprogramming level = 2)

For large scale machines we implemented and tested three barriers on the KSR:

Tree – Mellor-Crummey and Scott's tree barrier with flag wakeup [MCS91a]. This algorithm associates processes with nodes (both internal and leaves) in a static 4-ary fan-in tree. After waiting for their children (if any) and signaling their parent (if any) in the arrival tree, processes spin on locally-cached copies of a single, global wakeup flag. The last arriving process sets this flag. On KSR's ring-based topology, the resulting invalidations and reloads approximate hardware broadcast.

Com-tree – A competitive variant of the Tree barrier, in which processes spin for only a bounded amount of time, as in the C-sp-blk algorithm of the previous section.

Scal-SC – A scalable scheduler-conscious barrier that uses the C-sched barrier among the processes on a given processor and a scalable tree barrier across processors. This is the final algorithm presented in Section 3.4.3.

Barriers based on a centralized counter do not scale well to larger machines for two reasons. First, their critical path length is linear in the number of processors; second, the centralized counter can become a significant source of contention. Given that processes do not migrate among processors, the Scal-SC algorithm avoids these problems while making optimal spin versus block decisions.

Figure 3.21 compares the performance of the various barriers in our synthetic application on the KSR 1, with a multiprogramming level of 2, and with varying numbers of processors. Repartitioning decisions were made at one-second intervals. As can be seen from the graph, the Tree barrier is rendered useless with the introduction of multiprogramming. Its performance degrades due to the large number of context switches required in order to go through a barrier episode, and the amount of time wasted before each context switch—equal to the scheduling quantum. It is surprising to see that even the "spin then block" heuristic of the Com-tree barrier performs quite badly in the presence of multiprogramming. While processes do not have to waste a quantum before yielding their processors they still have to suffer the large number of context switches that degrade performance. The Scal-SC barrier improves performance by an order of magnitude compared to the Com-tree barrier. It requires the minimum possible number of context-switches, while still maintaining the logarithmic path length and low contention properties of the Tree barrier.

As we mentioned in Section 3.4, the Scal-SC barrier can be sensitive to the frequency of scheduling decisions. We ran experiments to determine the level of sensitivity. Figure 3.22 shows that if the time between repartition decisions is very small, performance degrades quite sharply. We believe, however, that repartitioning will be a rare event on large machines—as rare as the arrival and departure of jobs from the system. For repartition intervals greater than 500 ms, the Scal-SC barrier performs well.

To validate the synthetic results, we ran a barrier-based version of Gaussian elimination on 57 KSR processors.[14] The results appear in Figures 3.23 and 3.24. In the absence of multiprogramming the Scal-SC barrier is only slightly worse than the Tree barrier, and significantly better than the Com-tree barrier. The introduction of multiprogramming renders the Tree barrier useless; its performance degrades by at least an order of magnitude. At the same time, the Scal-SC barrier outperforms Com-tree by more than 50%.

3.6 Synchronization Conclusions

In this chapter we presented solutions to the problem of synchronization on multiprogrammed multiprocessors, for both small and large-scale machines. The mechanisms we presented are suitable for real-time applications, such as the lock used in shepherding or the barrier used in robot potential following algorithms. The mechanisms also apply to a broader range of parallel applications, such as the locks used in Cholesky or the barrier in SOR. Pseudo-code for the various algorithms remain in an anonymous ftp site at the University of Rochester and are currently in /pub/packages/sched_conscious_synch.

[14]We used pthreads to express parallelism in our barrier experiments. Due to limitations in the pthreads environment on the KSR only 57 of the 64 processors in the partition could be utilized.

We identified the main sources of performance loss for the two most common types of synchronization algorithms: locks and barriers. We also demonstrated that the scalable versions of synchronization algorithms based on distributed data structures are particularly sensitive to multiprogramming. Using a slightly-extended kernel interface, in which processes are able to defer preemption briefly, we examined several heuristic techniques—preemption-safe `test_and_set` locks (developed by previous researchers), "handshaking" queue and ticket locks, and competitive spin-then-block barriers—that avoid the worst performance anomalies in multiprogrammed systems.

We then proceeded to define an extended kernel interface allowing communication of process state information between the user and the kernel. We used this interface to construct *scheduler-conscious* algorithms: the Smart Queue mutual exclusion and reader-writer locks, the Scheduler Information small-scale barrier, and the scheduler-conscious scalable barrier. We demonstrated that these algorithms perform well in the absence of multiprogramming and provide significant performance advantages over their scheduler-oblivious counterparts in a multiprogrammed environment.

For barrier-based applications, the Scheduler Information barrier and the scheduler-conscious scalable barrier clearly outperform both the heuristic, spin-then-block barriers and the scheduler-oblivious alternatives. For lock-based applications the choice between preemption-safe centralized (`test_and_set` or ticket) locks and scheduler-conscious queue locks is less clear. Increasing the multiprogramming level decreases the contention observed by the application, since the number of processes accessing a synchronization variable concurrently is reduced. As a result, scheduler-conscious queue locks were often (though not always) inferior to the centralized alternatives in our experiments. As future increases in machine size increase the number of contending processors in multiprogrammed environments, the balance should tip back toward queue-based algorithms. Moreover, it is likely that coherence protocols on future machines will lack the ability to efficiently keep track of a large number of processors sharing a common variable. As a result, the cost of coherence management for the data structures of centralized synchronization algorithms is likely to be unacceptably high. This will again argue in favor of queue-based algorithms, in which no two processes spin on the same location.

4 Scheduling

Scheduling involves deciding when to run a given task, how much time to allocate to it, and in a multiprocessing enironment where to run it. The scheduling discipline has long formed the core research work in the hard real-time community. In a hard real-time application it is imperative that a schedulability guarantee be provided to the user, as well as the conditions under which the user can expect a set of tasks to be scheduled. Scheduling mechanisms are typically set up to handle specific assumptions about the types of tasks to be considered, e.g., regular and periodic may mean tasks need to run every five seconds. Each scheduling discipline is targeted for a class of applications exhibiting a set of definable constraints and properties. The goal of this chapter is to describe and evaluate a set of scheduling paradigms appropriate for SPARTA environments.

We have described many properties of SPARTAs in past chapters. In this chapter we focus on the fact that SPARTAs are programs responding to real-world events and that the processes that generate schedulable work in the real world can be modeled by continuous functions. Further, in many SPARTA applications, the derivative of execution time (approximated by the change in execution time from run to run) of those functions is small. We have developed three scheduling policies particularly suitable for use in SPARTAs: Derivative Worst case (DW), Standard Deviation (SD), and Last One (L1). DW allocates time based on a maximum increase over the last execution time; SD allocates time based on a certain number (definable) of standard deviations over the mean execution time for a sliding window of times; and L1 allocates time based on the last execution. In addition we provide a standard Worst Case (WC) scheduling algorithm. In keeping with the Ephor philosophy, each task can be scheduled using a different policy. These policies provide different levels of guarantees and require different amounts of user information.

The chart in Figure 4.1 shows a comparison between the different scheduling mechanisms. It indicates if they require user input, if they provide guarantees, whether they are adaptive, and their relative performance. In this chapter we show that the DW policy provides superior performance for tasks whose time dis-

Policy	Require User Input?	Provide Guarantees?	Adaptive?	Proc Utilization Application Perf
WC	Yes	Yes	No	Poor
DW	Yes	Yes	Yes	Decent
SD	No	Statistical	Yes	Good
L1	No	Statistical	Yes	Very Good

Figure 4.1: Comparison of the scheduling mechanisms

tribution varies slowly. For those tasks, DW obtains better processor utilization than the WC policy while still providing absolute guarantees. L1 achieves highest processor utilization, but at the cost of occasionally missing deadlines. SD provides approximately the same utilization as DW without requiring (perhaps tedious) timings from the user, but provides only statistical guarantees. All of L1, SD, and DW are more suitable than WC for SPARTA environments. In the rest of this chapter we describe each of the policies in greater detail, provide insight into the mechanisms used to implement them, and provide an evaluation of their performance.

4.1 Policies

The Worst Case (WC) policy is taken from the hard real-time community. There are many different implementable versions based on variations of the classic rate monotonic work [LL73]. These variations account for different models of the tasks such as regularity or periodicity. The worst case policy states that the user provides the system with the longest time a particular task will ever require. The system scheduler allocates time based on this worst-case value regardless of whether the tasks are statically (once at initialization) or dynamically (throughout execution) scheduled.

The drawbacks of this approach are, first, that the user must provide the worst-case time (if the user fails in this endeavor then the system scheduling guarantees no longer hold), and second and more serious, is the fact that this worst-case time required to be provided by the user may only occur in a small fraction of the parameter space of real world functioning, i.e., for the (vast) majority of times the task is executed, its time may be significantly less than its worst-case time, and thus valuable processor cycles will be wasted. In reality, this is a frequently occurring phenomenon. The strength of using worst-case times though is that the scheduler can provide the absolute guarantees that are needed in some environments.

While some soft real-time system designers have begun ignoring worst-case timings because of the huge waste of resources, we believe that in many applications, including SPARTAs, certain portions will desire guarantees similar to those provided for hard real-time applications. We therefore investigated mechanisms that would take advantage of the properties of SPARTAs and yet be able to provide the guarantees being made in the hard real-time sector. This work produced two policies: Derivative Worst case (DW) and Standard Deviation (SD). These two policies share the goal of providing guarantees to the SPARTA programmer that are more favorable than those provided using WC. They differ in the type of guarantee provided and amount of user provided information required.

The Derivative Worst case policy (DW) allocates time for a task based on the amount of time taken the last time the task ran plus a (user provided) percentage of that last execution time. The goal of the DW policy is to provide absolute guarantees to the user like WC while achieving considerably improved processor performance. It accomplishes this by requiring the user to provide the system with the maximum possible amount (percentage) of change between any two execution times. In this way the system can guarantee that enough cpu time is allocated to that task. The DW policy will work extremely well in environments that change slowly over time. For example, consider a processing task with time proportional to the number of objects in the world, e.g., cars on a city grid map. It is unlikely that in one instant there would be zero cars in the grid and the next instant a thousand, rather it is more likely that over time some cars will leave and some will enter with the net effect at each step being a small increase or small decrease. An observation of the DW policy worth noting is that the time can drop sharply without causing a problem. It is only the maximum rate of *increase* that is of concern. There are imaginable scenarios where the DW policy would perform worse than the WC policy. If a task running time was almost always near its worst-case time or if it had very sharp increases then the WC policy would be better. However, for many classes of tasks the DW policy will perform considerably better than WC. The DW policy shares the strength of the WC policy of providing absolute guarantees, but also shares one of its weaknesses of requiring tedious or potentially difficult to obtain information to be provided by the user. Its primary strength is that it achieves good processor utilization while still providing scheduling guarantees.

The Standard Deviation policy (SD) derives its name from the fact that it allocates time based on a z-value (number of standard deviations above or below the mean) of execution times collected over a sliding window of previous executions. Both the z-value and the size of the window are user definable. The default is a z-value of one and a window of ten. The amount of time allocated to a given task based on the default paramenters is the mean time over the last ten execution times plus one standard deviation of the last ten execution times. Given a model of the task execution time spread and the deadline constraints of the application, the user can choose an appropriate z-value. A strength of the SD policy is that it does not require the user to provide any information (in empirical tests the default values perform well). Another strength is that it allows analysis of the expected deadline miss rate for a known (or assumed) distribution of execution times. Its drawbacks are that it provides only statistical (and not absolute) guarantees, and that tasks with erratic execution times a high z-value may need to be chosen.

The Last One (L1) policy allocates time for a task based on what the last execution time for the task plus a small percentage to account for timing error, scheduling overhead, etc. This is the simplest of all policies requiring no information from the user and making no guarantees about deadlines. In reality this policy works very well because there are two places where slack (extra) time occurs in creating a schedule. One is that there is time left over after all the tasks that will fit on a processor have been assigned. And the second is that some of the scheduled tasks may take less time than they did last execution. These two sources of slack time provide sufficient "breathing room" for this to be an effective policy. Its strengths are that it provides the highest processor utilization and requires no information from the user. Its drawbacks are that it provides no guarantees. Under some execution time models and for some parameters one or more of the scheduling mechanisms will be identical. If all execution times are equal then L1, SD, and DW and all equivalent. This is true since the standard deviation and derivative worst case increase will both be 0 thus collapsing to the L1 policy. For a distribution of monotonically decreasing execution times L1 is equivalent to DW since the derivative worst-case increase can be set to 0. As the standard deviation of a distribution of execution times goes to 0, SD will tend toward L1.

4.2 Analysis

Analyzing scheduling policies in real-time systems often involves determining the schedulability of a policy given a set of assumptions, or ascertaining the policy that provides the best schedulability for a given set of assumptions. However, in our work, the analysis needs to proceed along slightly different lines. We assume that there are always enough processors to schedule the hard and periodic tasks. Our concern is *how many* processors are left available for the soft real-time and high-level parallel tasks. The more the better.

To capture this behavior we define a number called processor utilization. This is the amount of time available on a processor divided by the amount of time actually used. For example, if there was one task scheduled on a processor, and every second of wall time that task executed for a half of a second, then processor utilization would be 50 percent or 0.50. Notice it does not matter how much time was scheduled for the task, it could have been scheduled for a half a second, the full second, or any time in between. Only actual execution time matters in calculating processor utilization. Therefore, given any distribution of execution times (but constant), the WC policy will have lower processor utilization than the L1 policy. This is because when the scheduler allocates time under the WC policy it needs to allow for the maximum possible execution time. In all but the worst case, this will waste potential. It also means fewer tasks can be scheduled on each processor because each requires a larger allocated block of time. The fewer tasks (assuming they execute in the same time regardless of the scheduler) will have a lower summed execution time and consequently lower processor utilization. We have thus captured the desired effect. The processor utilization provides a hard metric to compare scheduling policies along one dimension. Processor utilization will be more thoroughly examined in Section 4.4.

Another means for comparison is how frequently a task misses its deadline, or how likely such an occurrence is. As is standardly defined, a missed deadline occurs when a task has not completed by the time it needs to run again. This would occur in an average case scheduling paradigm if a task takes longer to complete than expected and all other tasks execute in the expected amount of time. This situation would prevent the first task from having access to the processor again by the beginning of its next scheduled cycle. Since we assume that there are enough processors to schedule all the tasks, WC and DW will never (in theory) miss any deadlines, because they always leave enough room available for any circumstance (as defined by the user). It is therefore never a system or runtime fault should a deadline be missed under either of these scheduling policies, rather it would be the user's fault for not entering the correct worst-case time (WC) or derivative worst-case increment (DW).

However, using SD or L1 it is possible, even expected (with low and defined probability), that occasionally a task will miss a deadline. In the rest of this section, we will analyze the probability with which this may occur. A missed deadline under L1 or SD will occur when the policy underestimates the amount of time needed for the tasks of this execution cycle. Recall L1 derives this estimate from the last execution time and SD from a given z-val above the mean over some previous window of execution times. If many tasks are scheduled on the same processor, then the probability of missing a deadline is not based on one task, but rather on the interaction of all tasks scheduled on that processor. For example, it is possible for one task to be over its expected execution time, another task to be under, and the net execution time to be within the scheduled time.

To analyze the probability that a given set of tasks takes longer than the amount of time allotted by a scheduling policy we assume that the distribution of times is Gaussian. This assumption appears reasonable as shown by the representative graph in Figure 4.2 of the execution times from the vision processing task of the shepherding application.

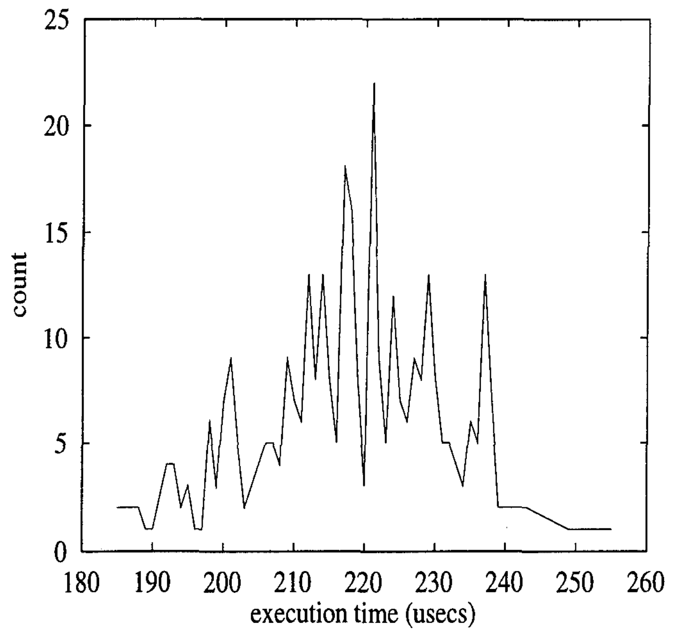

Figure 4.2: Execution times for the vision processing task

We define I to be the scheduling interval allotted to all the tasks. In a straight L1 policy I is assigned to be μ_e and we arrive at a model as shown in Figure 4.3 with probability density function:

$$P(x < z) = \int_{-\infty}^{z} \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{1}{2}[(x-\mu)/\sigma]^2} dx.$$

To determine the probability that a deadline is missed, we need to know how likely it is that the sum of the execution times is greater than the scheduling interval I. Given that $x \equiv$ some event from the sum of the times of scheduled tasks, then $P(Miss_x) = P(x > \mu_x)$ where

$$\mu_x = \sum_{i=1}^{n} \frac{\mu_i}{n}.$$

For the rest of this discussion, we will, for convenience, set μ_x to be 0. Therefore the probability under these assumptions that we miss a deadline is

$$1 - \frac{1}{\sqrt{2\pi\sigma}} \int_{-\infty}^{0} e^{-\frac{1}{2}[x/\sigma]^2} dx,$$

which easily simplifies to 1/2.

In reality, however, Ephor schedules in a fixed extra amount of time, call it δ, to allow for timing inaccuracies, scheduling overhead, calculation error, etc, as well as extra "breathing room" (note you could trade efficiency for safety by modifying δ).

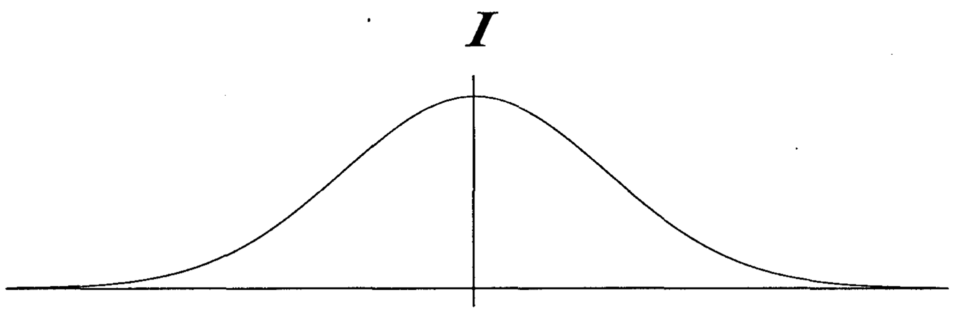

Figure 4.3: Gaussian distribution with I set assuming no offset and no slack

The model shown in Figure 4.4 represents the model with δ included. Here again I represents where the interval is set. Notice that in this case we have a smaller probability of missing a deadline. Specifically the $P(Miss_x) = P(x > \mu_x + \delta)$. Solving for the PDF gives us $1 - \int_{-\infty}^{(\delta-\mu)/\sigma} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}[(x-\mu)/\sigma]^2} dx$, and again setting μ_e to 0 gives us

$$P(Miss_x) = 1 - \frac{1}{\sqrt{2\pi}\sigma} \int_{-\infty}^{\delta/\sigma} e^{-\frac{1}{2}[x/\sigma]^2} dx,$$

which can be evaluated using tables of the normal probability distribution found in most statistics books.

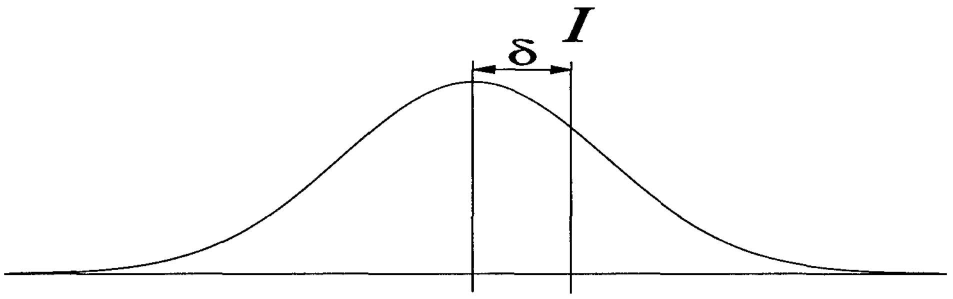

Figure 4.4: Gaussian distribution with an offset but no slack

Both of the above models have assumed that the tasks are able to be "packed" on to the processor in a perfect fit. However, even after placing as many tasks on a processor as possible, there is still going to be *slack* time left over due to the inability to find a perfect fit. As before, if we assume a random mix of jobs, the slack time will tend toward a Gaussian distribution. Again, to get a feel as to whether a Gaussian distribution is a reasonable model, we collected sets of slack times for a different numbers of tasks and for varying times. Figure 4.6 shows the distribution when there were few tasks involved in being scheduled, and Figure 4.5 illustrates the slack time distribution when many tasks are involved. The complete model is now represented in Figure 4.7 and when these two random variables are summed, the distribution of the result is a Gaussian distribution as shown in Figure 4.8.

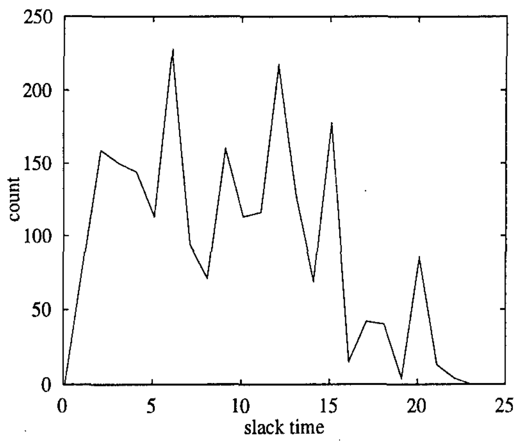

Figure 4.5: Slack times for many tasks

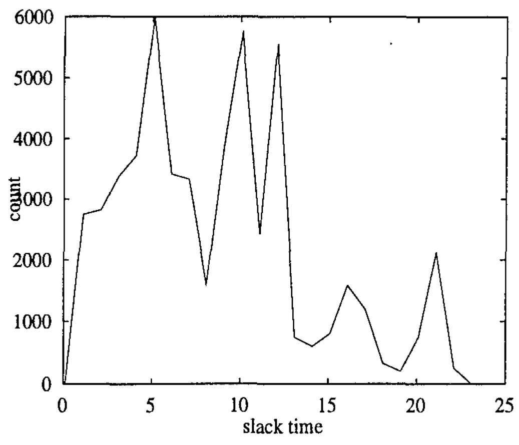

Figure 4.6: Slack times for few tasks

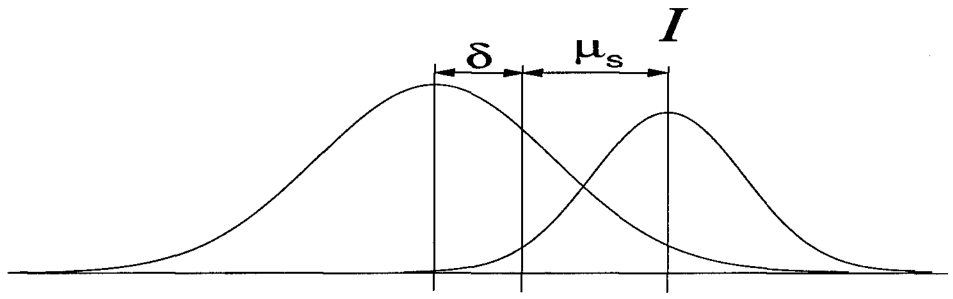

Figure 4.7: Gaussian distribution for execution and slack times

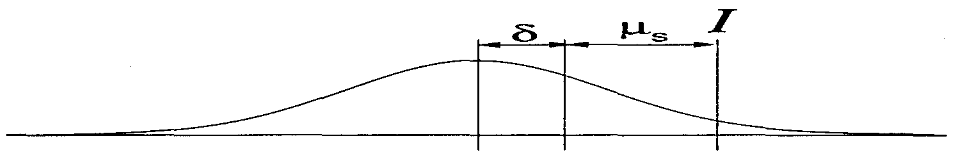

Figure 4.8: Summed Gaussian distribution with offset slack

Here we want to know the probability such that

$$P(Miss_x) = P(x > z_e) \bigwedge P(x < z_s),$$

where value$_s$ are values taken from the slack distribution. Since the number of packable tasks can be given by $\lfloor \frac{I-\delta}{\alpha} \rfloor$ where α is the mean of the distribution of slack times. Then

$$slacktime = x - \delta - \lfloor \frac{x-\delta}{\alpha} \rfloor$$

and the PDF can be evaluated as

$$P(Miss_x) = \int_{x=0}^{\infty} P(x > z_e) \cap P(x < z_s).$$

Since $P(A+B) = P(A)P(B)$, continuing along this line of reasoning would lead to a convolution integral. However, we know from probability theory that $\mu(x-s) = \mu_x - \mu_s$ and $\sigma^2(x-s) = \sigma_e^2 + \sigma_s^2$, and we know that the convolution of two Gaussians distribution is a Gaussian distribution. Thus substituting into our last solved PDF with $\mu_e = 0$ we get

$$P(Miss_x) = 1 - \int_{-\infty}^{(\delta+\mu_s)/\sqrt{\sigma_e^2+\sigma_s^2}} \frac{1}{\sqrt{2\pi\sqrt{\sigma_e^2+\sigma_s^2}}} e^{-\frac{1}{2}[x/\sqrt{\sigma_e^2+\sigma_s^2}]^2} dx,$$

which again can be evaluated using most statistics books. This gives us a description of how likely each scheduling policy is to miss a deadline based on a certain input set of tasks. The remaining question is how well the different policies *perform*. We address this question in Section 4.4. With these two pieces of information the tradeoffs for the tasks in the application can be evaluated and the appropriate scheduling mechanisms can be requested from Ephor.

4.2.1 Time Distributions

For the previous analysis we have assumed a Gaussian distribution of execution times. In addition to the overall distribution the time pattern of execution times is very important. To provide a better understanding of the scheduling policies we describe the effect the time distribution can have on the policies. For the graphs in this section wall time will be on the X axis and execution time on the Y axis. For example a task with five execution times of 10, 5, 10, 15, and 10, appears in Figure 4.9, Section A. The scheduling policies will handle most time distributions adequately and as expected, but some distributions will have a large effect on their performance. For example the time distribution in Section B (of Figure 4.9) will cause problems for L1, SD, and DW. L1 and SD will miss half the deadlines, while DW would have to specify a very large derivative. A monotonically decreasing distribution as in Section C is ideal for all policies except for WC, which will have low processor utilization, while a monotonically increasing distribution as in Section D could be a particular problem for L1. For such a distribution δ should be increased. These time distributions were just made as examples, and we would expect real-world time distributions to be closer to the one in Section E. The closer the time distribution is to section E the closer the actual behavior will be to the behavior analyzed in this section or the empirical behavior presented in Section 4.4.

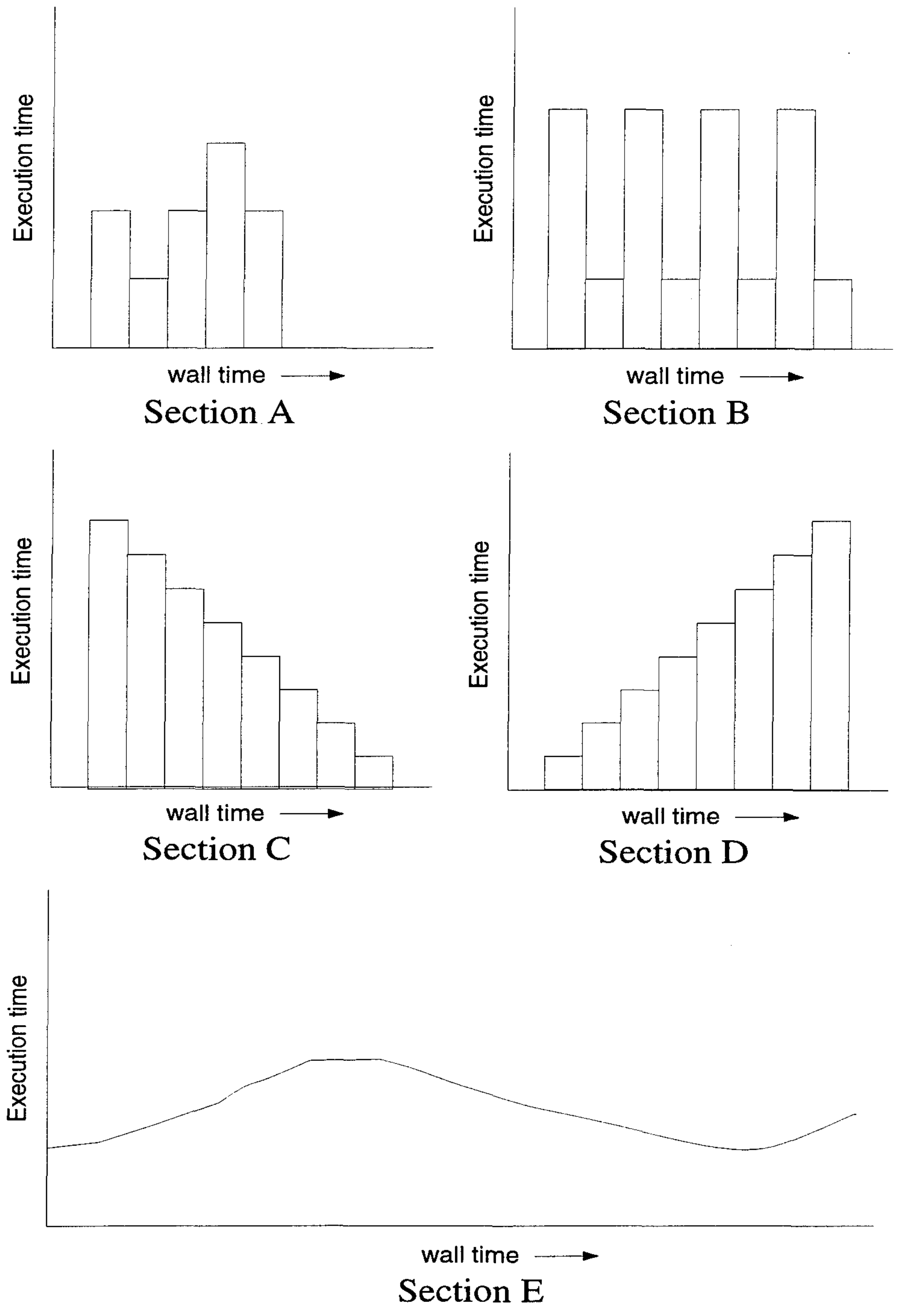

Figure 4.9: Time distribution examples

4.3 Implementation and Mechanisms

There are several different pieces required to turn the different scheduling policies into actual Ephor mechanisms. First, is a method provided to allow the user to request the desired policy and to provide any information required by that policy (worst-case time for WC, worst-case increment for DW, and number of standard deviations (z-value) and window size for SD). As with any scheduling mechanism, the correctness of WC and DW can be guaranteed if and only if the user has entered worse case times or increments that hold true for the duration of the application. The mechanism for specifying this information is provided by the Ephor startup functions. For convenience and flexibility, Ephor allows the user to change the scheduling policy dynamically for a given task using a set of runtime functions similar to the initialization calls. It is also worth noting that different policies can be chosen for different tasks. Thus, a scheduling policy can be chosen that is best suited for each task. Care should be taken however when mixing policies for periodic soft real-time jobs. While tasks with hard real-time priorities are separated from other periodic soft real-time tasks, there is no distinction made among soft real-time tasks. It is possible then, if tasks with DW and L1 are mixed, that one of the DW tasks might miss a deadline because an L1 task ran over. So while it is possible to set the scheduling policy for a given task individually, it is important to understand the interactions between the tasks.

As mentioned in Section 4.1 the L1 and SD are "hands off" for the user, requiring no input of potentially difficult to obtain information. This does, however, place a greater burden on Ephor to insure that a reasonable schedule is built. Ephor needs to track the time each of the tasks took to execute. This is accomplished by a header and footer that is inserted into every task. Among other things, the header marks the time a task starts and the footer marks when the task finished. The 21ns granularity bus cycle counter is used and provides ample resolution for timing any task in our system. Since this is a multiprocessing environment, it is possible for races to occur between the tasks writing their execution time and the scheduler or Ephor reading it. In an actual implementation the kernel would have control at the beginning and end of each task and this race would not be an issue. However, since we employed a user-level scheduler, we needed to "coordinate" with the tasks. One solution would be to introduce locking. However, this would have introduced undesired overhead. Instead, for L1, a median value of the last three timings are used. In the rare event that a timing was garbled it posed no real problem. Serendipitously, using the median value also simplified other problems such as timer wrap. In implementing the SD policy a similar behavior was achieved by tossing out the highest and lowest times within the window of observation.

The different scheduling policies affect how the requested time slot for each task will be determined. If a task has requested the WC policy then the time slot requested of the scheduler for that task will always be what the user has explicitly specified as the worst-case running time for that task. For the DW policy the requested time slot will be the last time plus the worst-case increment based on the last time (here too it is actually the median of the last three times). For SD it will be the default number of standards above the mean. The number of deviations above the mean can be changed by the mean can be changed by user if so desired. Finally, for L1 it will request the median of the last three running times.

4.3.1 Scheduling Algorithm

At the point the scheduler is given a set of tasks, it is no longer aware of what policy generated the time slot for each task. All it is presented with is a set of tasks and associated with each task is a priority, a period, and a requested time slot for that task. The scheduler's goal is to pack the tasks on as few processors as possible while not violating any constraints, such as combining hard and soft real-time tasks or overbooking processors. The scheduler sorts tasks first by priority and then by period. It uses a multiprocessor version of the standard rate monotonic scheduling algorithm. It places as many of the smallest period tasks as possible on the lowest numbered available processor (some have been reserved for special functions), and then the next, etc. Ephor does not attempt to schedule to 100 percent processor utilization but instead schedules to a fixed amount short (previously defined as δ) of 100 percent to allow for timing inaccuracies, scheduling overhead, calculation error, etc, as well as extra "breathing room"(currently about five percent). For the next sorted group it again starts with the lowest numbered available processor. It cycles through all the available processors for each bin of tasks in the sorted list. An important side effect of this algorithm is that task migration is infrequent and occurs only when the cumulative execution time of the tasks have changed such that a task no longer fits its previously assigned processor, or it can fit on an earlier one. As part of this process the scheduler needs to keep track of the processor utilization and can make this information available through Ephor to the application. Ephor also makes available the last execution time of a task and a running average over the selected SD window.

4.3.2 Methodology

As with the other Ephor experiments, the results in this chapter are from our twelve processor SGI Challenge machine with 100MHZ R4400 processors. To isolate the effects of the scheduling algorithms completely, all non-essential (standard Unix processing, graphics, etc.) computation, including the scheduler itself, was shipped to the remaining processors. In this way we guarantee the observed effects are strictly based on the differences between the different scheduling policies.

The scheduling algorithms had eight available processors on which to schedule the tasks.

In addition to the requisite tasks for the shepherding application, we created a set of tasks for which we could vary: the number that existed, the time each task took to execute, and the period. To allow for fine control of the machine load this information could be set individually for each task. For each scheduling algorithm, the tasks, both the shepherding and the controlled tasks, were placed onto processors as described above in implementation section 4.3. The goal was to place these periodic and predictable tasks onto the fewest number of processors leaving as many processors as possible available for the high-level parallel planner. The assumption being that given more processors, and thus more processing power, the parallel planner would be able to arrive at a better solution in the given time constraints. The ability to "pack" the tasks onto processors was affected by the scheduling algorithm used.

There were two classes of information we desired from our experiments. One was the application performance. This metric simply involves counting the number of confined sheep after the system has reached stabilization. Results reported are from a representative run. This is an application-dependent metric but provides insight into how much an application might benefit from using one particular scheduling policy versus another. This number is easily obtainable and required no special instrumentation for these experiments. The larger the number of sheep confined the better the application's performance is considered to be.

The second and more important metric is processor utilization. Processor utilization allows an application independent comparison of the scheduling policies. Higher processor utilization is considered more desirable since this meant a scheduling policy is better able to use the resources of the system. Processor utilization is defined as the actual amount of time the tasks took divided by the amount of time scheduled for them. For example, if a policy dictated that a task be given a two millisecond block fifty times a second but the task took only one millisecond every time it ran, then processor utilization would be fifty percent. A processor utilization is computed independently for each processor and an average processor utilization is computed for all the processors required to run the given set of tasks. For example, if a scheduling algorithm required two processors to schedule all the tasks while another required four, then the second algorithm would have a utilization half that of the first. The most interesting number then is average processor utilization, as this implicitly contains the number of processors required for a given task set. Remember the fewer the better. Note that it is possible for the processor utilization metric to be greater than one hundred percent. This can occur if the tasks on a given processor take more time to run then is allocated to them. This phenomenon is obviously not desired because it means some task had to miss a deadline. In fact, this is one empirical number we can apply to a policy like L1 to provide insight into how often tasks suffer (miss a deadline) because of its attempted average case performance.

Unlike the number of sheep confined, processor utilization values were not readily available. To obtain them, timing code was inserted into the header and footer of each task. The code made use of the fine-grain 21 nanosecond clock available on Challenge architectures. This resolution is sufficiently fine grain to time any of the tasks used in our application. The execution time of the task was divided by the rate of the task and summed across all tasks. This value represents the percentage of actual cpu use for a given block of time on a given processor and is defined to be the single processor utilization. To obtain average processor utilization, all individual processor utilizations were summed and then the sum was divided by the number of processors required by the scheduling algorithm.

We were able to vary a number of parameters independently, including the number of control load tasks, the amount of time these tasks took, the period of the tasks, and their priority. The cross product of varying all these parameters across experiments is very large. We explored the parameter space and provide results from the portions highlighting an interesting comparison between different scheduling policies in sections 4.4.2 and 4.4.3. Other portions of the space, while quantitatively different, represent the same qualitative results.

To create a controlled environment for exploring the parameter space we fixed the shepherding application so the actuators had no effect on the real world. We call these experiments the synthetic ones meaning they did not perform any real work, or more accurately the effect of their work did not have an impact on the real world. Instead we made sheep escape the field at a constant rate. In so doing we precisely controlled how much load was generated by the application. These experiments are useful for showing us the interesting parts of the parameter space. For example, in some regions (of the parameter space), we determined, as reported in Section 4.4.2, there was no difference between the different applications primarily because the machine was so under-loaded that even a very poor policy was sufficient. From our experience, and in interaction with other researchers, we do not expect this to be the case for real applications. In fact the opposite is more often true, i.e., the machines are over-loaded.

To further control the amount of load in the system we created additional tasks. The execution times of these tasks could be controlled by parameters. We could make them constant or vary with the number of sheep in the field (as many of the actual tasks for shepherding do). The ability to vary the number of tasks allowed us to explore the policies when handling anywhere from a few tasks (about 5 - minimum needed to make the simulator work) to a large number of tasks (about 50)

All the experiments were started with a fixed number of sheep placed in random configurations (position and direction) around the field. Over time some of the sheep escaped the field. The remaining ones were considered confined. In the shepherding simulations, the application reaches a stabilization point where it is able to maintain a fixed number of sheep without allowing any more to escape. This was the number used in the application performance comparisons.

4.4 Performance and Results

The results in this section are taken from our shepherding simulator. It is a real-time simulator, not a simulator of a real-time application: The processor simulating the real world is not affected by other work in the system. Thus the SPARTA can fall behind the (simulated) real world.

The performance of a real-time application is often correlated with how efficiently it can use the resources on the machine. Indeed, the results in this section show a correlation between high processor utilization and good application performance. Achieving higher processor utilization involves a tradeoff either in providing different information about the tasks, or in accepting the possibility that on occasion a task may miss its deadline. In this section we will show that L1 has the highest utilization, but a task using the L1 scheduling paradigm is more likely to miss a deadline than if using any other paradigm. SD has the next best utilization but may not be appropriate for some (highly variable (in execution time) tasks) and still admits missing deadlines. DW is close to SD, but it guarantees no deadlines are missed. It too may not be effective for highly variable tasks. In addition it requires user input where L1 and SD do not. It does, however, achieve much better processor utilization and performance than WC while maintaining the same guarantees as WC. As expected the WC policy translates into the worst utilization and performance.

As a reminder, we used two metrics to evaluate the different scheduling mechanisms, one application dependent and the other application independent. The application metric is the number of sheep confined within the field after stabilization, and the application independent metric is the processor utilization. The number of sheep confined is given in a bar chart where the number for a given scheduling algorithm represent the number of sheep confined at the stabilization point. The processor utilization results are presented as graphs with time on the x axis and processor utilization (between 0 and 1) on the y axis. The sheep confined charts are straight forward, but it is worth highlighting the salient features in the processor utilization graphs. The experiment we ran was to start with 350 sheep moving in random directions. Many tasks (such as the vision processing or perimeter checking) are tied to the number of sheep on the field, so over time the work required of these tasks diminishes.

4.4.1 Graph Interpretation

There are two classes of utilization graphs: individual processor utilization graphs and average processor utilization graphs. For each experiment there are eight different individual processor utilization graphs and one average processor utilization graph. The average graph is generated by summing up all processor utilization on active processors and dividing by the number of active processors. There is considerable information contained in the processor utilization graphs. Figure 4.10 shows a processor utilization graph for a synthetically generated program for processor 7 from a run with many(50) tasks. Remember that the graphs will look very smooth and regular for the synthetic runs since they represent very controlled settings. The first thing to observe is that for processor 7, utilization drops to 0 at around 25 seconds. This indicates that all tasks have been assigned to other (lower numbered) processors, a good thing because that now means this processor is free and available for running a high level planner, such as the parallel save sheep planner. Another interesting feature is the step-like nature of the graph. In the first 20 seconds each vertical step is an increase in processor utilization. This occurs because as the amount of work to do in the application decreased (fewer sheep surviving to monitor) an extra task could be taken from processor 8 and assigned to processor 7 thus increasing the effective workload on processor 7. While the processor utilization on 7 shows a step increase, at this instant the average processor utilization would show no change. The time from approximately 20 to 25 seconds represents the period where tasks were taken off of processor 7 and assigned to processor 6. Notice also between 20 and 25 seconds there is lag from when L1 and SD move a process to when DW does. This is indicated in figure 4.10 by the lag denotation. At around 25 seconds we would expect to see an increase in average processor utilization since now the scheduler has managed to pack the tasks onto one fewer processors and processor 7 is no longer required. The processor utilization for WC is a steadily decreasing function for our experiments, because as sheep leave the field and there is less actual work required to monitor the remaining ones, WC continues to allocate the maximum needed time for that task.

Next we examine the processor utilization from processor 4. This is the first processor the scheduler attempts to place tasks on, so it will always have some processor utilization. Again for illustration purpose we have taken the results from the synthetic shepherding application. There are three prominent features in Figure 4.11. The steady decrease in the curves followed by the instantaneous rise is caused by the fact that as sheep escape there is less work needed to monitor the remaining ones. As the work becomes small enough, it becomes possible to place more tasks onto processor 4 and an increase in processor utilization occurs. The size of the steps becomes smaller over time because the time of each of the tasks in the application is decreasing, thus, a smaller drop in available cycles is more

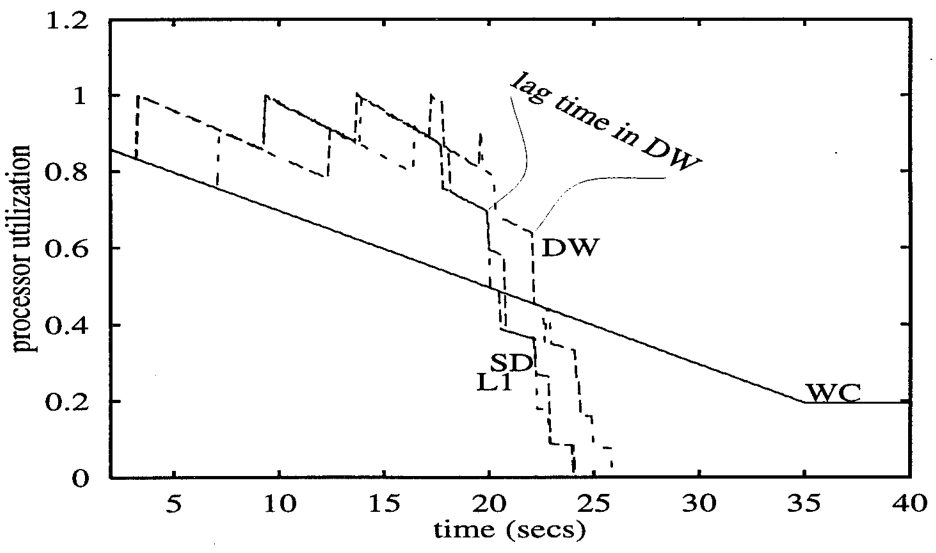

Figure 4.10: Example processor 7 utilization

readily filled by some available task from another processor, and consequently migrating that task causes a smaller gain in processor utilization. Again the smooth decrease in the WC plot occurs because the amount of scheduled time for the tasks can never decrease, yet the actual execution time does.

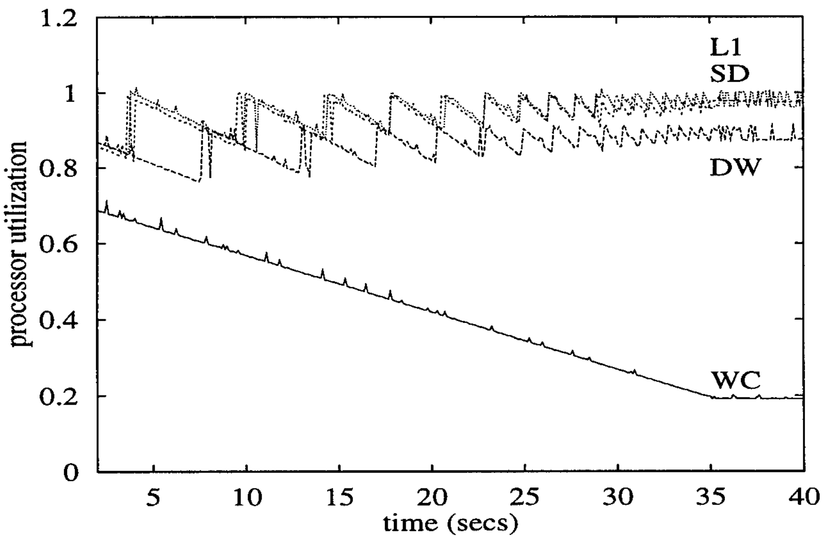

Figure 4.11: Example processor 4 utilization

The last illustrative example shown in Figure 4.12 represents the average processor utilization from the same experiment as the utilizations for processor 4 and 7 were taken. As alluded to in the previous examples, the spikes in this graph occur when all the tasks can be packed onto one fewer processors. At that time there is the same amount of actual work occurring, but on one fewer processors, thus the average processor utilization is higher. Again the WC utilization is lower. Also observe (as indicated in the figure by the lag denotation) the increased amount of time taken by DW before moving to a higher processor utilization. With the ability to interpret the processor utilization graphs we turn our attention towards results from the different experiments that were performed, synthetic and real, and their significance.

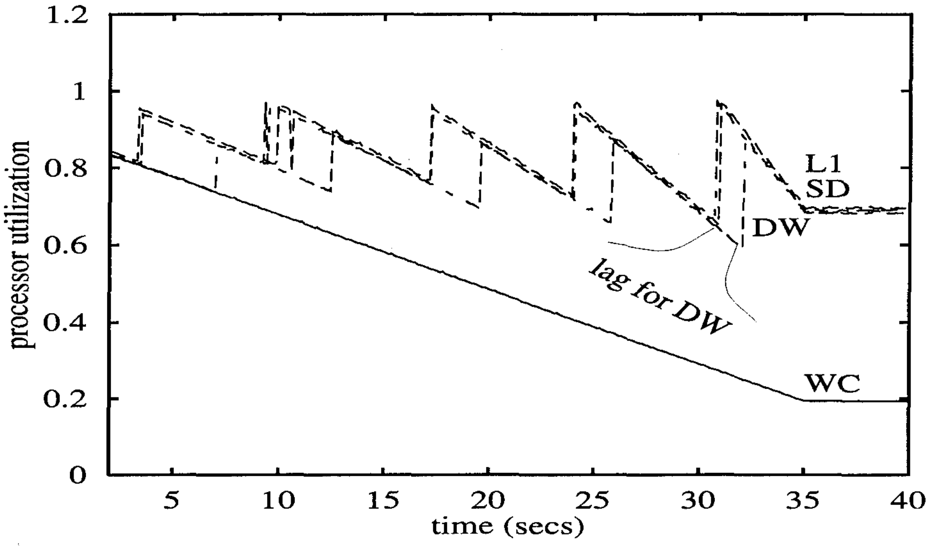

Figure 4.12: Example average processor utilization

4.4.2 Results from Synthetic Runs

When there is very low load placed on the machine by the application, there is not much difference in performance between the different policies. The amount of load is determined by the amount of computation per task. Low load represents minimal computation, high load represents the maximal computation, and moderate load is between the two. Figures 4.13 and 4.14 show the average processor utilization under low load conditions for few tasks and many tasks. The utilization graphs are essentially the same since when there is very little for each task to do, the sum of their execution times is such that they can all be placed on one processor.

As additional load is placed on the system the differences between policies begins to show. Figure 4.15 shows a comparison average processor utilization for the different mechanisms running under moderate load. Even under moderate load WC is wasting resources. Also under moderate load DW is starting to separate from SD and L1 as indicated by the lag in decreasing the number of required

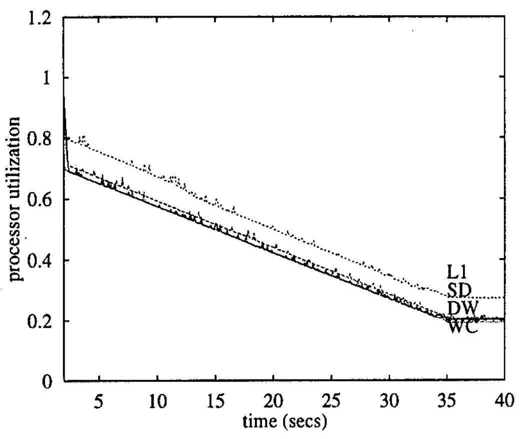

Figure 4.13: Average processor utilization with few tasks and low load

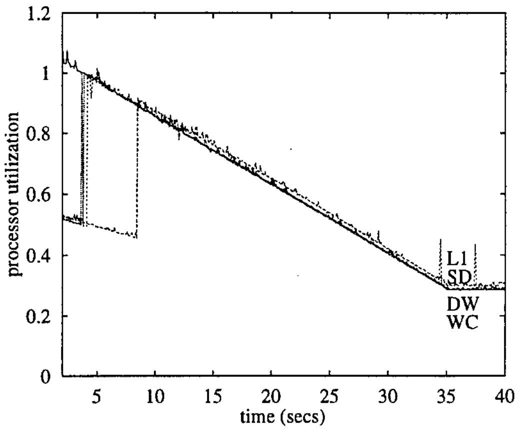

Figure 4.14: Average processor utilization with many tasks and low load

processors. A more telling sign of this effect is shown in figure 4.16, where on processor 5 we see more clearly the lag in the time required (by DW) to move the tasks off of processor 5.

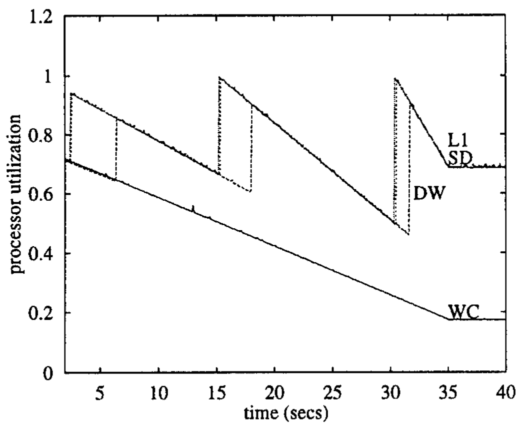

Figure 4.15: Average processor utilization with moderate tasks and low load

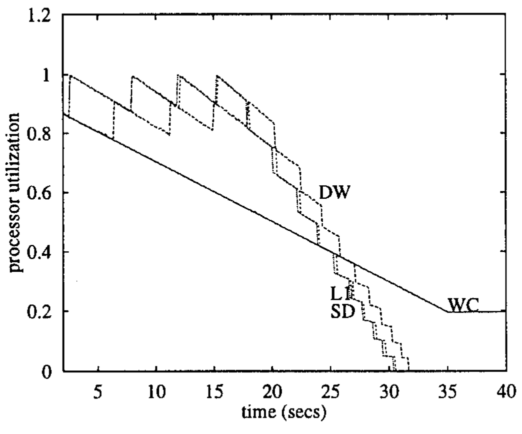

Figure 4.16: Processor 5 utilization with moderate tasks and low load

The most interesting and important portions of the parameter space is the region where the number of tasks and amount of computation per task cause a high load to be placed on the machine. Figures 4.17, 4.18, and 4.19 show the results when a high load is placed on the parallel machine for few (5), moderate (25), and many (50) tasks. Under high load and a large number of tasks the DW mechanism shows lower processor utilization for significant portions of the time. The reason SD and L1 are nearly identical is because for the synthetic programs the variance in execution time is very low. This also allows DW to perform better than otherwise expected. In Section 4.4.3 the performance between these three policies is differentiated. The final synthetic experiment we ran was to change the rate at which the tasks ran to see how this affected performance. Figure 4.20 shows an experiment similar in amount of work to 4.19, but with a greatly increased rate (10 times) on the controlled tasks. The behavior for this and other different rate experiments is similar to the default rate for the same amount of work.

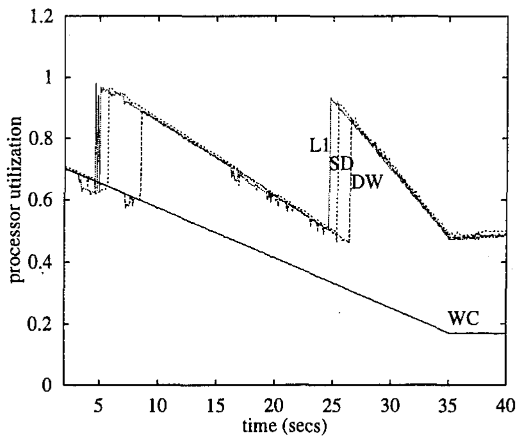

Figure 4.17: Average processor utilization with few tasks and high load

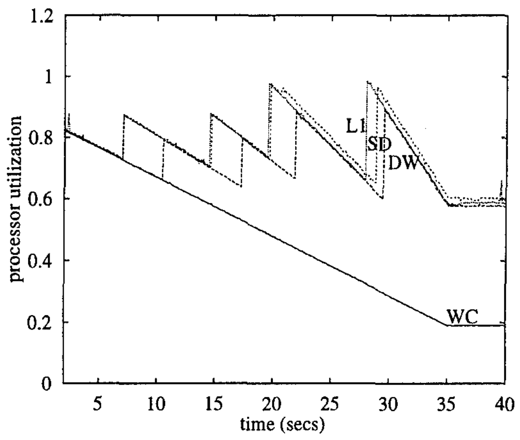

Figure 4.18: Average processor utilization with moderate tasks and low load

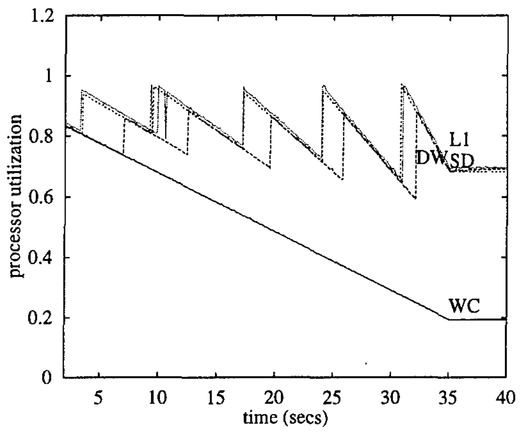

Figure 4.19: Average processor utilization with many tasks and high load

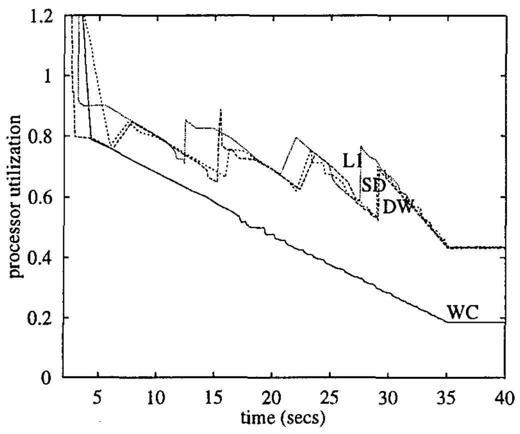

Figure 4.20: Average processor utilization with many tasks and high load increased period

4.4.3 Results from Real Program

We report results from two different scenarios of running the actual real-time shepherding simulator. In the first one, only tasks required by the shepherding application were scheduled. In the second version we added 15 additional tasks whose execution time distribution was similar to that of the shepherding tasks. The motivation for this second experiment was to get a feel for how the mechanisms would perform if they had to handle a larger number of tasks. Figure 4.21 shows average processor utilization results from the first scenario and Figure 4.22 from the second scenario. We see that now the mechanisms' behavior is quite different. As expected, L1 is maintaining high processor utilization, and WC has poor utilization. The mechanism based on the SD policy achieves second best processor utilization and WC next. Figures 4.23 through 4.26 show the processor utilization of individual processors 4 through 7 for the second scenario.

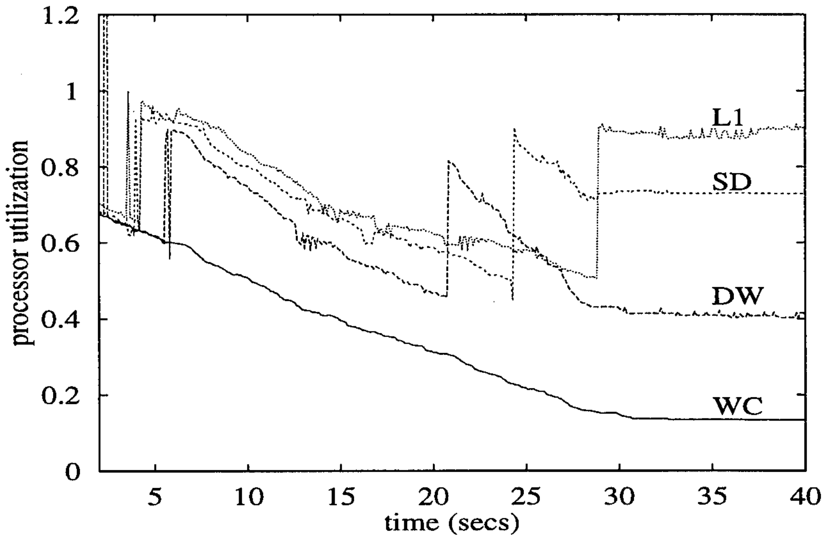

Figure 4.21: Average processor utilization from real program with few tasks

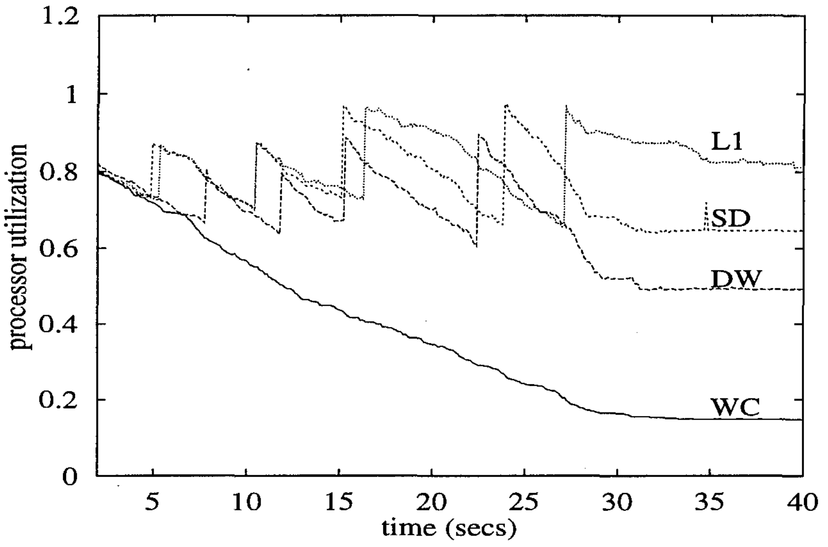

Figure 4.22: Average processor utilization from real program with many tasks

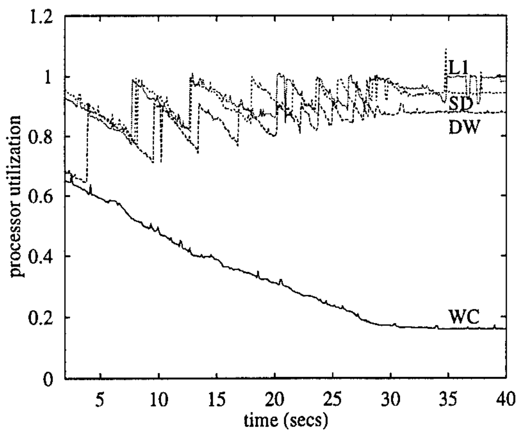

Figure 4.23: Processor 4 utilization for real program with many tasks

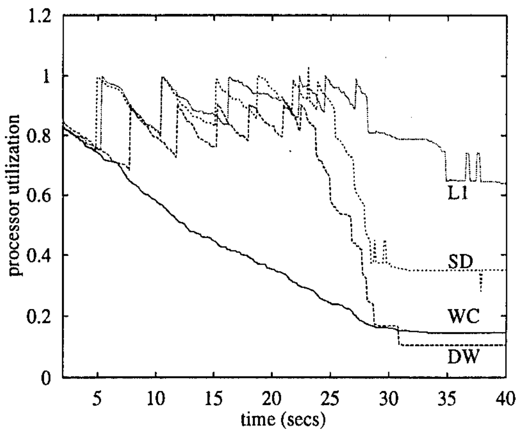

Figure 4.24: Processor 5 utilization for real program with many tasks

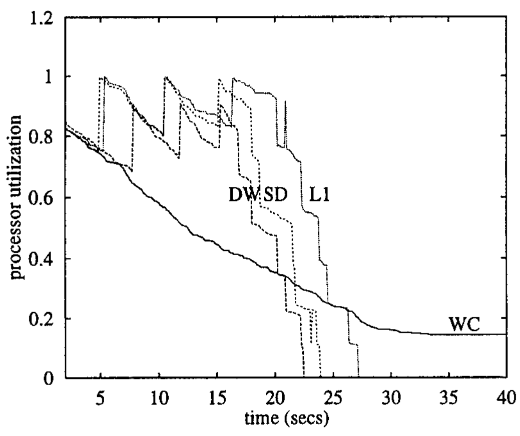

Figure 4.25: Processor 6 utilization for real program with many tasks

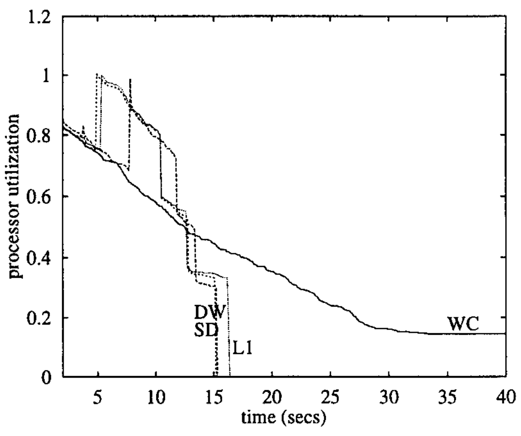

Figure 4.26: Processor 7 utilization for real program with many tasks

DW provides the same scheduling guarantees as WC, so if the user can provide the worst-case increment much better utilization can be achieved while still meeting guarantee requirements. DW outperforms WC for many SPARTA applications since their behavior is based on real-world events and the real-world tends to be continuous without instantaneous changes. Thus WC is an ideal candidate for real-world applications requiring guarantees. For applications with a greater tolerance to missed deadlines, an approach like SD or L1 will achieve better processor utilization and will most likely translate into better application performance as shown in Figure 4.27 (for the first scenario) and Figure 4.28 (for the second scenario)

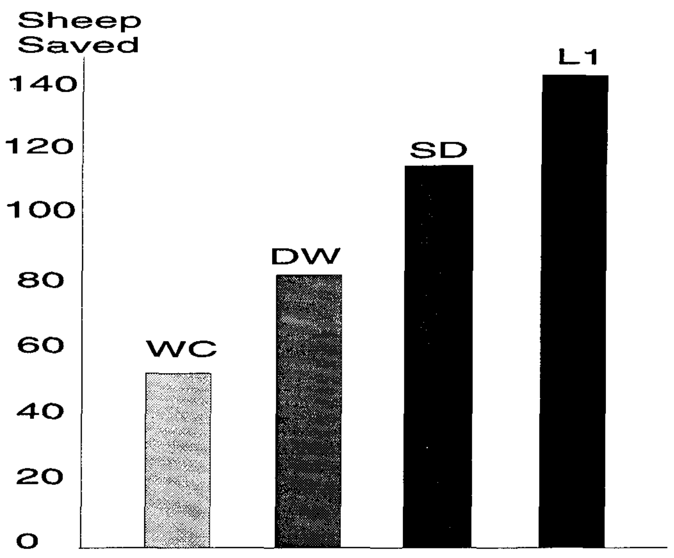

Figure 4.27: Number of sheep confined with few tasks in system

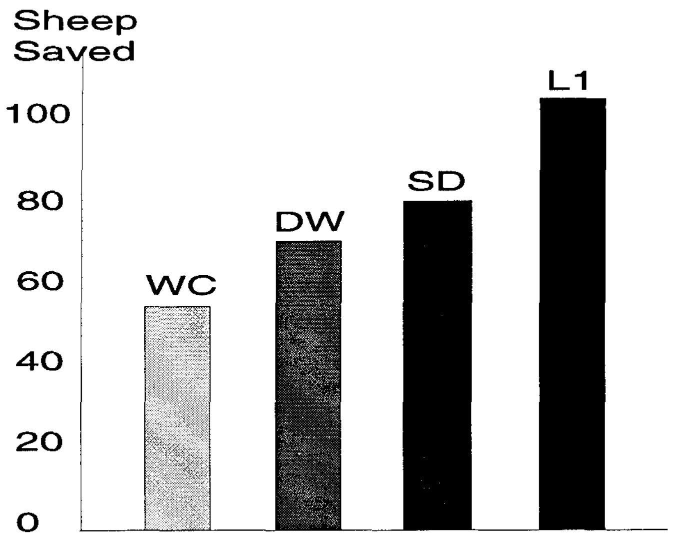

Figure 4.28: Number of sheep confined with many tasks in system

4.5 Scheduling Conclusions

The results reported in this chapter from the application independent processor utilization metric, and from the application dependent application performance metric, allow several conclusions to be drawn:

- The **last one** mechanism provides the highest processor utilization among the mechanisms examined. For applications that can afford to miss a deadline occasionally, this translates into high application performance.
- For applications that have a strict deadline requirement and can provide worst-case increments, the **derivative worst case** mechanism will significantly outperform the **worst case** mechanism.
- For applications that want a controlled probability of missing a deadline the **standard deviation** mechanism provides a good alternative to the less precise **last one** mechanism, while still providing better performance than either of the mechanisms based on worst-case times or increments.
- It is possible to select different mechanisms for different tasks thereby choosing the mechanism most suitable for a given task. Users should be careful though about the interactions of the different mechanisms since a less strict policy such as L1 or SD could compromise a strict one such as DW or WC.
- One of the benefits of the **last one** and **standard deviation** mechanisms is that they do not require any user input. This property is well matched to the Ephor philosophy of having the runtime take responsibility for managing resources from the user. These policies also provide better performance than the worst-case mechanisms.

5 Ephor Mechanisms

In this chapter we present the set of mechanisms available from Ephor. This set includes some new mechanisms we have designed as well as some other successful mechanisms that have been incorporated from the literature. The set of mechanisms described and evaluated in this chapter provides a reasonably comprehensive set (enumerated in Section 5.1) of what a SPARTA programmer would require when designing an application. Additionally, Ephor is extensible, that is, if another mechanism is desired by a particular SPARTA programmer, it could be incorporated into Ephor fairly easily. Ideally the user definable mechanisms would be linked in at runtime. In Section 5.1 we provide a list and description of each of the mechanisms. Section 5.2 discusses dynamic technique selection in more detail and provides performance results. In Section 5.3 we reiterate the usefulness of an adaptive scheduling paradigm and show its effectiveness when combined with parallel process control. We finish the chapter in Section 5.4 with conclusions from our examination of mechanisms.

5.1 A Set of Mechanisms

The mechanisms we have included in Ephor allow the SPARTA programmer to concentrate on the control of the application rather than on resource allocation issues. The mechanisms focus on adaptability, providing the user flexibility to adapt to a changing world. In this section we describe each mechanism and give examples of its use.

1) Dynamically select technique based on internal system state.

Ephor keeps track of resource allocation. When a task needs to run, Ephor checks to see the different possible techniques the application has provided for this task, and based on the current resource utilization choses the *best* one. In defining the techniques the user provides an ordering. The *best* one is the best (user defined) technique that will finish given the current processor load and resource allocation. For example, suppose there are two different techniques the user has provided for executing the save_sheep task. The first runs quickly and produces a satisfactory choice and the other computes for longer but produces a much better choice of the next sheep to save. At the moment the task needs to be run, Ephor will know whether there is time available to run the longer task, and if so, will run it. If there is very high load on the system at the time the task needs to run, Ephor will automatically run the quicker task so it and other tasks do not start missing deadlines. In some real-time applications, such as factory resource allocation tasks, the goodness value for completing a task decreases over time, so in addition to the trade-off of completion there is expected completion time. If this function was input to Ephor, then in addition to or instead of, trading off whether it would get completed, it could trade off against this diminishing goodness value. As another more qualitative example consider a case where there are two sensors on a mobile robot for determining the distance to a wall, an infrared sensor (quick) and a binocular set of cameras (slow). If, at the moment the robot requests the distance_to_wall task be run, the infrared sensors are busy tracking a moving object, then Ephor can automatically run the binocular vision task to get the requested distance. The beneficial part about this is that the application obtains the desired information without having to track the resource allocation of the distance sensors and realize that another portion of the application was using them. The mechanism is more thoroughly explored in Section 5.2.

2) Schedule tasks using derivative worst case or adaptive scheduling policies.

The scheduler provided with Ephor is discussed at length in Chapter 4. The key idea is to provide scheduling policies based on observations about real-world applications. Specifically, Chapter 4 recommends a derivative worst-case scheduling algorithm for programs needing to provide absolute guarantees. This significantly outperforms the standard worst-case scheduling for many real-world tasks. The chapter further argues for using an adaptive scheduling policy such as one based on statistics of execution times over a previous window. These provide much higher processor utilization and that frequently translates into better application performance. In Chapter 4 we analyze the distributions of execution times and quantify the tradeoffs between the different policies.

3) Dynamically control the placement and quantity of parallel processes.

The idea behind dynamic parallel process control is that an application will go through periods of lower demand and periods of higher demand based on the current state of the environment. For a task that can be run in parallel and adjust the number of processors it uses (see Chapter 2), Ephor can dynamically increase or decrease the number of processors available to that task. Used in conjunction with adaptive scheduling mechanisms, allowing the number of tasks to vary can yield large performance gains. Without the ability to dynamically vary the number of processors available to a task the worst-case situation would have to be chosen to apply for all time. Choosing the worst case might be especially detrimental if the worst case is based on the infrequent event of several different parallel techniques running simultaneously. If, however, Ephor can adjust the processors assigned to the given task, then when the different parallel tasks need to run concurrently Ephor can assign each fewer processors, and when (say in the normal case) they run separately, Ephor can allocate to them more processors allowing them to complete their execution either quicker or better or both. This mechanism is explored more in Section 5.3.

4) De-schedule all running subtasks associated with a particular task.

Ephor is in a unique position to handle semantically grouped subtasks. This is because at initialization the application informs Ephor of its program structure. In particular, it indicates all the subtasks that comprise each technique associated with a task. If an application determines it longer needs the results of a task, then Ephor can then de-schedule all the subtasks associated with that task. For example, consider the shepherding application. If after the application requests a solution to save_sheep a wolf appears on the field, the subtasks associated with save_sheep would no longer be needed. In fact they would interfere with the task needed to respond to the wolf. Because the solution to the save_sheep task is time dependent (based on the position of the sheep and shepherd at the time the task was started), there is no reason to suspend the subtasks; if possible, they should be eliminated. Since Ephor is aware of all the subtasks associated with the save_sheep task, it can perform this operation.

5) Automatically time tasks and update their status block.

Ephor needs to time tasks to perform many of its operations such as scheduling or dynamic technique selection. The times are gathered by reading the fine-grain 21ns timer and updating a time value in the header and footer of each task. This information can be valuable to the application, and Ephor makes it available via Ephor function calls. In many real-time systems, time information flows in the other direction, that is, many real-time systems require time as an input. Ephor, however, *provides* times for the different tasks to the user. This is in keeping with Ephor's philosophy of removing from the SPARTA programmer as much of the responsibility of managing resources as possible.

6) Provide access functions to share information between the runtime and application.

Ephor needs to track considerable information about resource allocation in order to perform many of its functions. It also needs to be aware of certain aspects of the application. It cooperates with the kernel and performs some system functions (scheduling) thus acting as a bridge between the application and the underlying system. This communication of information is accomplished via a set of access functions. These access functions allow the application to inform Ephor of its program structure, to request scheduling policies for its tasks, to override the standard dynamic technique selection mechanism, and more. The access functions also allow information to be communicated from Ephor to the application. The application may request the time for a given task, whether (and to what task) a particular resource is currently being used, as well as current processor utilization. The application can use this information to guide its decisions about how to interact with the environment. For example, if the application was contemplating placing the program in a state of greater demand, perhaps by moving to a more complex area of the environment, it could check the current processor utilization and see if there was enough available utilization to perform the expected tasks.

Below is a list of the available functions from Ephor. The first three are special in that they take a variable number of arguments terminated by a NULL. To provide an idea of the possible arguments we give a typical initialization call. For the rest of the functions we provide their library definition. Following the list is a description of the functions.

```
ephor_hl_task_t
ephor_create_hl_task(ephor_periodic, True,
                     priority, 0,
                     ephor_rate, 16666,
                     NULL)

ephor_technique_t
ephor_create_technique(hl_task_number,
                       ephor_cpu_time, 16666,
                       ephor_resource_t, my_resource,
                       NULL)

ephor_task_t
ephor_create_task(hl_task_number, technique_number,
                  ephor_function, funct_name,
                  ephor_term_time, NEVER,
                  NULL)

ephor_resource_t
ephor_create_resource()

ephor_boolean_t
ephor_resource_in_use(ephor_resource_t resource)

void
ephor_run_hl_task(int hl_task_numb)

void
```

```
ephor_kill_hl_task(ephor_hl_task_t hl_task_numb)

ephor_sched_pol_t
ephor_set_sched_policy(ephor_sched_pol_t policy)

ephor_sched_pol_t
ephor_set_hl_task_sched_policy(ephor_sched_pol_t policy,
                               ephor_hl_task_t hl_task_numb)

void
ephor_overdemand_handler(ephor_hl_task_t hl_task_numb)

ephor_time_t
ephor_get_technique_time(ephor_technique_t, technique_number)

ephor_time_t
ephor_get_hl_task_time(ephor_hl_task_t, hl_task_numb)
```

Upon initialization the user informs Ephor of its program structure via three calls: ephor_create_hl_task, ephor_create_technique, and ephor_create_task. The arguments to ephor_create_hl_task allow the user to specify whether the task is periodic (Ephor will automatically schedule the task at the rate specified by ephor_rate), and its priority. The return from the function is a "cookie" that is passed to ephor_create_technique and used throughout the program when referring to this high-level task. The user can specify a worst-case time for a technique with the ephor_cpu_time argument. The user can also specify a list of resource with the ephor_resource_t argument. Note that Ephor expects a "handle" returned from ephor_create_resource(). The final call to establishing the program structure is ephor_create_task. This function expects cookies from both ephor_create_hl_task and ephor_create_technique, as well as a C function pointer following the argument ephor_function. The function also allows the user to specify an early termination time[1]. The final function used at initialization is ephor_create_resource. It returns a handle to a resource.

The rest of the functions are used throughout execution of the application and have the following functionality:

- ephor_resource_in_use returns a boolean indicating whether resource resource has been allocated,

[1]Currently this is an absolute time, which in some cases is not overly convenient. We envision that allowing a percentage would also be a useful feature.

- `ephor_run_hl_task` tells Ephor to execute `hl_task_numb`,
- `ephor_kill_hl_task` tells Ephor to kill `hl_task_numb`,
- `ephor_set_sched_policy` sets a default scheduling policy for all tasks,
- `ephor_set_hl_task_sched_policy` sets a scheduling policy for a specific task,
- `ephor_overdemand_handler` provides a handler for Ephor to run in case of over demand. If no handler is provided Ephor will take the default action described in **7)**,
- `ephor_get_technique_time` returns the time (L1) for `technique_number`
- `ephor_get_hl_task_time` returns the time (L1) for `hl_task_numb`

7) Detect and recover from overdemand.

Ephor detects overdemand when the amount of time needed to schedule the requested tasks exceeds what is available as determined by the scheduling algorithm described in Section 4.3.1. Note that this occurs *before* we have performed any iterations with the real-time kernel and received a "failed schedule" message. The ability to detect a failed schedule before submission is a contribution of Ephor and can be accomplished because Ephor is aware of the program structure. This avoids the possible problem of the application trying many combinations before realizing that even its minimum requirements will not be able to be satisfied. In the case of overdemand, one of several actions are taken. The first is an attempt to select less expensive techniques for the given set of tasks. If this does not rectify the problem - the cheapest technique for each requested task is still too expensive - Ephor will do two things. By default it starts removing the lowest priority tasks from the schedule until it is able to fit the tasks. It then passes this set of tasks to the kernel. If instead, the user has registered an overdemand task with Ephor, then Ephor clears the schedule and calls this task. The expectation is that the application will choose a different course of action and submit a less expensive set of tasks.

8) Allow early termination of tasks.

The idea for early termination of tasks is based on anytime algorithms or imprecise computations similar to Liu et. al. [LLS+91]. Certain algorithms produce a result after a certain minimum time and then continue improving on that result until finishing execution. Based on the load in the system a task based on this model may be halted after the point it has reached an answer though before it has completely finished execution. In a highly loaded system this allows other tasks to start running and make their deadlines. Ephor provides support for this model by terminating such tasks after the minimum time requested if there are other periodic tasks that need to be run or if there are other higher priority parallel tasks that could make advantage of the extra cycles that would be freed by termination of the anytime task.

9) Automatically allocate resources based on task priority.

If multiple tasks request a resource, Ephor will allocate that resource to the task with the highest priority and then will attempt to find another technique to execute the lower priority task. Among equal priority tasks Ephor will attempt to satisfy the most tasks. The optimal allocation algorithm is not polynomial so Ephor uses a heuristic where it marks any contended resources, attempts to assign other techniques where possible, and then greedily goes through and assigns the rest of the resources to tasks in order of increasing period. Again this removes the burden of handling resources from the user. As is true in most of what Ephor does the user can override the automatic process of selection with a directive that forces Ephor to use a particular technique with a given set of resources.

10) Provide user-conscious synchronization.

SPARTAs that contain parallel algorithms will need to synchronize. As Chapter 2 indicates to obtain high performance algorithms must allow processes to come and go throughout their execution. In Chapter 3 we explore the issues in providing synchronization algorithms that continue to perform well in the presence of multiprogramming. Next we discuss Ephor's dynamic technique selection and dynamic parallel process control mechanisms in greater detail.

5.2 Dynamic Technique Selection

5.2.1 Shepherding Simulator

As a reminder, the shepherding simulator is a real-time simulator, not a simulator of a real-time application: The processor simulating the real world is not affected by other work in the system. Thus the SPARTA can fall behind the (simulated) real world. In the simulator each sheep moves at constant velocity until herded, at which time it chooses a new direction from a uniform distribution whose mean is the center of the table with range plus or minus 45 degrees. The shepherd has a finite speed and can affect only one sheep at a time. The goal of the shepherd is to keep as many sheep on the table as possible. The performance of this SPARTA is defined as the number of sheep remaining on the table after having reached the steady state.

5.2.2 Planners and Choices

The important aspect of this portion of Ephor is how to select between different techniques for saving a sheep. We use six different combinations of planning methods and parameters, which, for clarity, we label as six different planners. The difference between the planners is how long they take to run and how many sheep are contained when running on an unloaded cpu. The expectation is that the more resources a task is allocated (including more cpu time) the better its result will be. Since we use a parallel machine we were able to dedicate one processor to the planning function, allowing accurate measurement of each planner's performance with no competing cpu load.

We measure time in simulator ticks. A sheep can travel from the center to the edge of the table in 100 ticks of the simulator and the shepherd can traverse the distance in about three and one half ticks. Planner A computes a list of all the sheep moving away from the table center that the shepherd has time to reach, sorted by distance from the center, planner A determines the best order for saving the top four sheep. The value for a particular ordering is calculated by a combination of number of sheep saved and projected time to save those sheep, with the number of sheep as a first-order variable and time as a second, i.e., if a particular ordering saves 2 sheep and another saves 3 then the second is always preferred. Among the orderings that save an equal number of sheep, preference is given to the ordering taking less time. Planner A performs the best under no load but takes the longest time to run (about 2 ticks).

Planner B simply tries to save the sheep farthest from the center. It runs much faster than A (about 1/80 of a tick) but does not perform nearly as well under no load since it is really reacting, not planning. The following difficulty indicates why B does not contain as many sheep as A: if by letting the farthest sheep go, it is possible to save the next two, A will save the two sheep while B will save only one. Planners C, E, and F are variants of A but differ in the amount of checking they do, such as whether the sheep is farthest from the center, or whether it can be saved. For example, planner C does not insure all the sheep placed on the list can be saved. Planners C, E, and F run in approximately 1/20 of a tick, 1 tick, and 1/10 of a tick respectively. Planner D is similar to planner B and runs in about 1/80 of a tick.

5.2.3 Experiments and Results

To evaluate the effectiveness of dynamic technique selection we compare the results of using it on several different planners. The planners were run planners under simulated conditions of parallel activity in other parts of the SPARTA whereby a controlled load was placed on the processor that the planner was running on. Since we were using a multiprocessor we could vary the load experienced

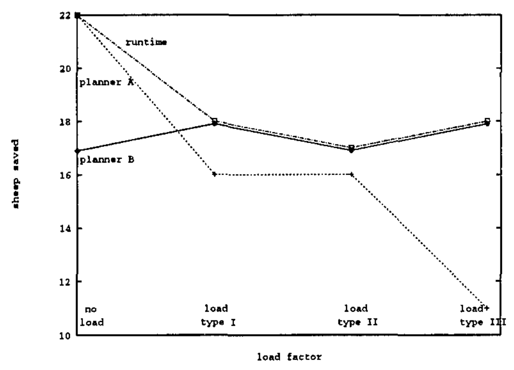

Figure 5.1: Adapting to high fixed loads.

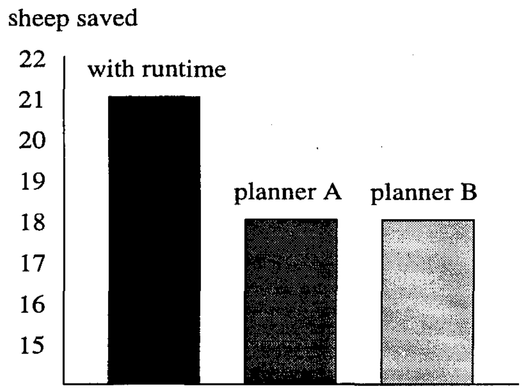

Figure 5.2: Adapting to high load, averaged variable loads.

by the planner process without affecting any of the other processes in the system. We also had tight control over how much load was experienced on the processor running the planners. The graphs in this section appear in pairs, with each pair representing an experiment. The line graph on the left always gives performance for a set of fixed loads, while the bar chart on the right gives average performance when the cpu load varies during the run.

The first experiment (Figures 5.1 and 5.2) demonstrates the effect of a high load. Recall that B runs about 160 times faster than A. Planner A should outperform B with no load, but more load could mean planner A might not complete its calculations in time, thus planner B should outperform A under high load. The loads are plotted on a logarithmic scale, that is **load type II** is twice as much as **load type I** and half as much as **load type III**. Indeed there is a dramatic decrease in performance of planner A under higher loads while planner B remains fairly constant. In a fixed load environment the runtime can choose between the better of the two planners, thus achieving the best performance in all cases. In the second half of the experiment the load varied through time. Half the time there was no load and half the time there was load. When there was a load it was divided evenly (by thirds) amongst the different load types. Figure 5.2 represents how A, B, and the runtime mixture perform under varying load.

The next experiment,(Figures 5.3 and 5.4) measures the effect of higher priority jobs running ahead the planners. This differs from the last experiment, in which there was equal probability of the planner or the load processes running. For this experiment we guaranteed that the load processes are always scheduled first. Recall planner D runs about 4 times faster than C. In the fixed load case we can see that each planner's performance degrades severely as the time taken by high priority tasks increases. However, the runtime is still able to select the better of the two. Under varying load the runtime performs almost as well as planner C and much better than D.

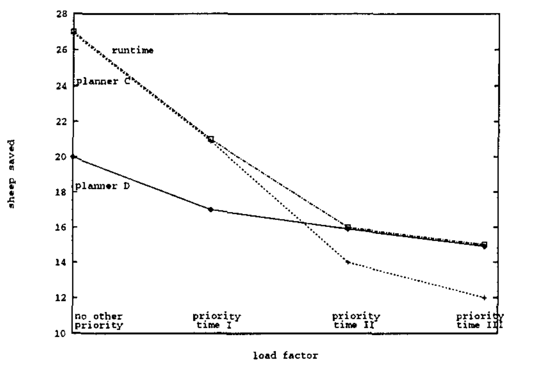

Figure 5.3: Adapting to high fixed priorities.

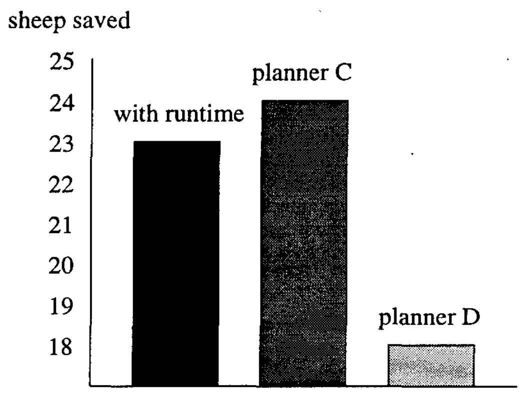

Figure 5.4: Adapting to high priorities, averaged variable priorities.

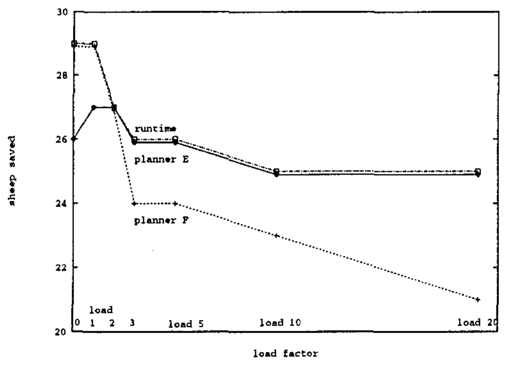

Figure 5.5: Fine adjustment between similar fixed loads.

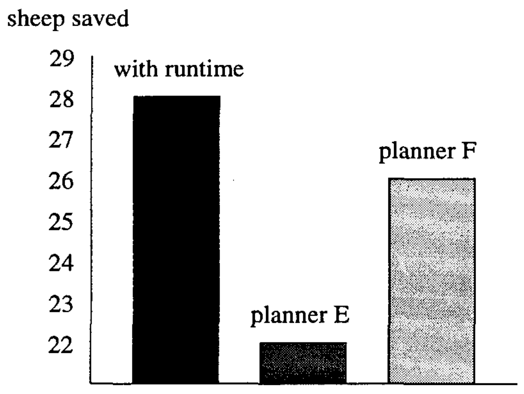

Figure 5.6: Fine adjustment between similar averaged loads.

The final experiment (Figures 5.5 and 5.6) is meant to approximate most closely a realistic application environment. This experiment measures the effect of a light to moderate load on the planner processor. We also expect the decision between planners will not be extreme, i.e., the planners we use, E and F differ by one order of magnitude in time requirements. For the fixed load case the runtime can always select the better of the two planners, yielding top performance. In the varying case, by dynamically selecting the appropriate planner, the runtime achieves higher performance than either of the planners.

5.3 Dynamic Parallel Process Control

Using dynamic parallel process control Ephor allows the application to take advantage of periods of lesser activity by allocating extra processors when they are available. As we described in Chapter 2 the application has to be designed to be able to take advantage of the situation. If it is, Ephor can help it use extra available processors. Many mechanisms need to come together to make parallel processor control effective. An adaptable scheduling mechanism should be employed, otherwise the same number of processors will always be available and the only difference will be whether Ephor needs to schedule multiple parallel planners at the same time.

To evaluate the usefulness of a planner and runtime that can dynamically vary the number of processors we designed a set of experiments and wrote a parallel planner that can dynamically adjust to the number of processors Ephor allocates to it. The goal of the planner was to determine the next sheep to save. The input to the planner is the current model of the world (sheep position and velocity) and the output is the next sheep to save. The farther the planner projects into the future, the better the choice will be. However, this future projection is a very expensive operation because we need to simulate sheep moves for an exponential number of possible orderings, and thus it requires tremendous computation. To design an adaptable parallel planner, we have a master compute all the different permutations of possible sheep n saves into the future. This operation is relatively cheap. Then, each of the slaves (the master can also act as a slave) computes a value for a possible ordering of sheep saves. If the value is less than a global minimum it updates (via obtaining a lock) the best current ordering and value.

We ran three sets of experiments to determine the potential benefit gained from being able to take advantage of additional processors when they become available. The difference between each of the experiments was the rate at which the sheep moved. Within each experiment we varied the maximum number of processors that would be available to the parallel task. At each available processor count we randomly chose the number of processors that would be available to run the parallel task this execution. To best simulate actual conditions, 50 percent of the time the maximum were made available, and the other 50 percent of time was divided evenly among the remaining possibilities. For example, the percentages used to obtain the application performance when 6 processors might be available to the parallel task would be 6 processors available 50 percent of the time with 5, 4, 3, 2, and 1 processors being available each 10 percent of the time. The results in Chapter 2, Section 2.4.2 are similar to these except taken only at one point (7 processors available) and without varying the number throughout execution. Figures 5.7 through 5.9 show the results of the experiments. Remember, the comparison is between a planner that cannot adjust the number of processes it uses and thus is locked into using only one versus a planner that can use more processors when they become available. The important aspect of this discussion is not that parallel planners (versus sequential ones) can improve the performance of applications, rather, that in real-world applications we need parallel planners that can dynamically vary the number of processors they use. Equally important, we need a runtime such as Ephor, that can support this desired behavior. The ability to adapt is most important in high demand situations as witnessed by the greater difference between the adaptable parallel planner and the fixed parallel planner in Figure 5.9 versus Figure 5.7. However, in all cases there is a significant improvement in performance for planners that have been written to use extra processes when they become available.

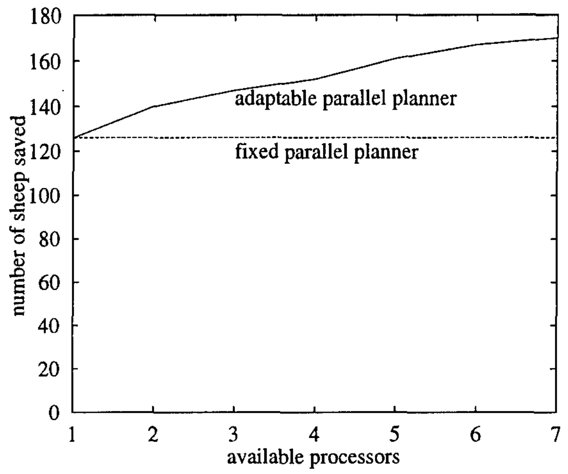

Figure 5.7: Fixed versus adaptable parallel planners: low demand.

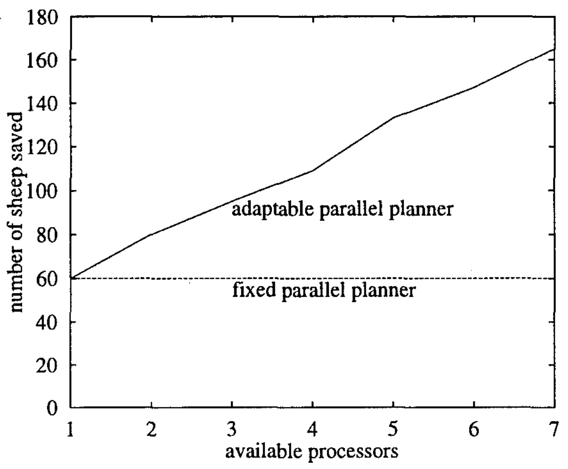

Figure 5.8: Fixed versus adaptable parallel planners: medium demand.

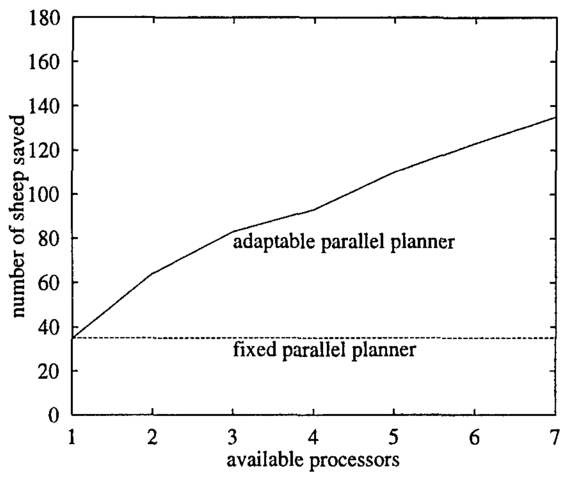

Figure 5.9: Fixed versus adaptable parallel planners: high demand.

5.4 Mechanisms Conclusions

The primary purpose of many of the mechanisms we described in this chapter was to provide flexibility to the application to allow the SPARTA to adapt to a dynamic world. We listed and described ten mechanisms that allow the system to be adaptable. In a previous chapter we examined in detail the scheduling mechanisms, and in this chapter we more thoroughly examined both dynamic technique selection and parallel process control showing the benefits of allowing programmers to have access to mechanisms designed to increase the adaptability of their application.

6 Shepherding: An Example Application

6.1 Introduction

We have indicated in previous chapters that the results presented were from our real-time shepherding simulator. This allowed us greater control in our experiments when testing the implemented mechanisms. Since it was a real-time simulator (not a simulation of a real-time application) it allowed us to explore real issues that occur in designing SPARTAs. However, designing real-world applications involves coordinating many pieces of hardware and integrating multiple software components. Thus, unforeseeable or hard-to-model issues often arise when incorporating real-world sensors and actuators into an application. For that reason while we developed Ephor (and our real-time simulator) we also implemented a real-world shepherding application in our robotics laboratory.

In this chapter, we describe the underlying hardware, including the camera, vision processing boards, processors, and puma robot arm. We then discuss the software components we designed to integrate the hardware components in real-time. At each stage we describe the trade-offs between the different possibilities and why the ones chosen were best suited for our environment. We also present results supporting our selection. At appropriate points we indicate underlying support provided by Ephor that eased and improved our implementation.

6.1.1 The Shepherding Application

The real-world implementation runs in our robotics laboratory [BB92]. As a reminder, it consists of self-propelled Lego vehicles "sheep" (see Figures 6.1 and 6.2) that move around the table "field" (see figure 6.21) in straight lines but random directions. Each sheep moves at constant velocity until herded by the robot arm ("shepherd"), at which time it is redirected back towards the center of the field. The shepherd has a finite speed and can affect only one sheep at a time. Figure 6.3 illustrates the different hardware components involved in the shepherding application. The goal of the shepherd is to keep as many sheep on the table as possible.

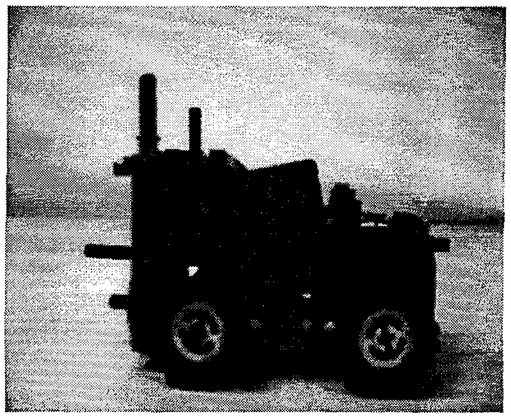
Figure 6.1: Side view of a Lego sheep.

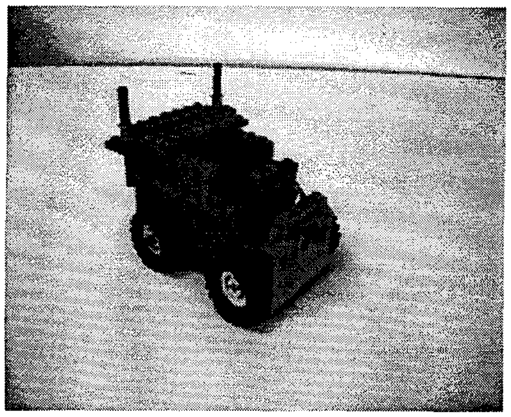
Figure 6.2: Top view of a Lego sheep.

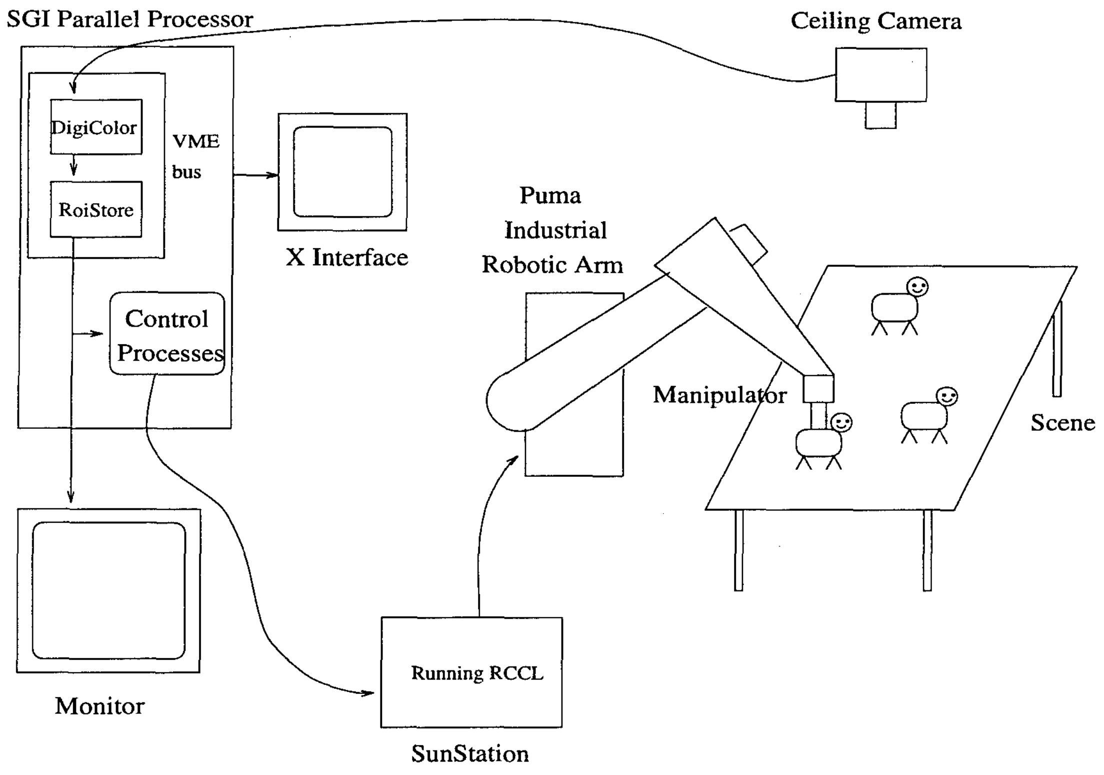

Figure 6.3: A flow diagram of the shepherding project.

The shepherding application is flexible and representative of a large class of applications. It includes high level cognitive models of the real world, planning, searching, sensing, acting, active perception, focus of attention, and multiple goals.

It contains situations in which overdemand can occur as well as the need for quick allocation and deallocation of resources. The shepherding application allows us to investigate many interesting properties of real-time systems that occur singly or in combination in other applications. Other real-world applications like navigation, game playing, laser tag, purposive vision, package delivery, and automated RSTA devices have properties similar to the shepherding application. In varying degrees all contain an element of search whereby the agent determines the next course of action. Most are designed around a high-level executive instructing lower levels to carry out actions. The executive reasons and makes decisions using a model of the real world. Its requested actions are interpreted by lower level's modules and translated before being carried out. Many contain an intermediate layer responsible for small corrections to the requested action (servoing). Many also contain a low-level survival layer whose actions need to be carried out constantly and can occur "subconsciously", i.e., without intervention from the higher levels. The shepherding application includes all of these properties. It uses various search algorithms to determine the next sheep to save. The planning code is a high-level executive working with models of the real world, sending out instructions to lower layers. The shepherding application's lower level interprets the requests from the executive (such as "deflect sheep at x,y"), and carries them out. An intermediate layer is employed to correct open-loop instructions by the executive. In the shepherding application this may mean fine adjustments to ensure the manipulator actually deflects the sheep even if the specified time or coordinates are slightly inaccurate. The shepherding application also contains low-level vision sensing that is constantly tracking the sheep. This occurs without the executive requesting it. Thus, the shepherding application embodies the important properties of many applications.

6.1.2 Real-Time Aspects

There are several crucial components involved in implementing the shepherding application: a real-time control component integrating the various modules, a vision module allowing tracking of the sheep, a communication module allowing the different machines to communicate across the ethernet, a manipulation module controlling the robot, and (for the user's convenience) a display module for debugging and tracking the application's progress.

The real-time control ensures each process runs at its specified intervals. In designing the modules to be used in the real-world shepherding special constraints needed to be satisfied. For example, often in the vision community separating an object from a scene could take several seconds per object, yet for real-time tracking we needed to track multiple objects many times a second. As another example, in the planning portions we needed to be able to place bounds on the time needed to make a decision. These examples provide insight into the difference between solving problems in real time versus off line. There are other requirements of the shepherding application, such as the need for high resolution object detection, that are discussed in detail in the appropriate sections.

6.1.3 Overview

Coordinating the different pieces of hardware with the software components requires an understanding of the requirements of both the underlying hardware and its associated libraries. Section 6.2 presents the details of the different hardware and associated software packages used to develop the real-world shepherding application. In Sections 6.3, 6.4, and 6.5 we carefully discuss implementation and tradeoffs of our application. Specifically, Section 6.3 presents the coordinating real-time component of the system and briefly describes the different processes and their functionality. Section 6.4 discusses the specific vision requirements needed by real-world shepherding, our solution to the problems, and the tradeoffs considered during implementation. We also discuss the algorithmic considerations of performing real-time multi-object object detection and tracking in Section 6.4. Section 6.5 concludes the implementation section discussing the difficulties and solutions of performing real-time manipulation. We discuss the capabilities of the application and make concluding remarks in Section 6.6.

6.2 Specifics of the Hardware and Software

This section contains detailed information about the hardware and software used to develop the shepherding application. The code segments that appear are not crucial in understanding the shepherding application, instead they are presented for the reader interested in specifics. They also illustrate additional difficulties encountered in real-world applications that do not exist in simulation. Since they are not extensive and fit well in the flow, they are in-lined in the appropriate section. The first two sections, 6.2.1 and 6.2.2, are specific to the twelve processor SGI Challenge Series that we attached the vision processing boards, and may be of interest only to readers with such platforms. The remaining section, 6.2.3, contains information on timings and board interconnections resulting from the configuration of Maxware boards we used.

6.2.1 Maxware Code Modifications

The Maxware software was originally written for a Sun workstation and had to be ported to the MIPS based SGI multiprocessor running IRIX. Below are the difficulties encountered during this port and the solutions we found. The solutions are the most straightforward to the problems that presented themselves and not necessarily the most elegant. It is also possible that some of the problems do not exist in current versions of IRIX, but these were the difficulties that arose when porting to IRIX 5.0.1.

The first set of problems encountered were the incompatibility of makefile formats. The makefile on the sun defaults to running */bin/sh*. IRIX defaults to running the shell of user executing the makefile. Thus the line "SHELL = */bin/sh*" was added to the top-level makefile. On our version of IRIX, the default compile was "ansii", but the software was not written with these conventions in mind. To handle this difficulty, a "-cckr" was added to all "CFLAGS" options. In many of the makefiles the "OFILES" definition was defined using the "FUNCS" definition, which was a combination of "CFUNCS" and "OBJSUF". When the makefile program was combining and substituting other definition to form "OFILES", it was not obtaining the proper list of object files. To solve this, the "OFILES" definition was defined by individually listing out each object file. When the makefile invoked ranlib numerous errors resulted. There was an incompatibility between the formats of the object files and what ranlib was expecting. After unsuccessfully deciphering the message, we simply added a empty executable ranlib to the end of our $PATH variable thus bypassing the creation of the table of contents for the archive.

There were other errors requiring only minor changes:

- In the *dcLutInit.c* file in the *dc* subdirectory **for** loops with the start condition of "i=-128" were modified to "i=(-128)"
- In the *mvAlloc.c* and *mvPage.c* files in the *mv* subdirectory the following lines were added to correctly define the "PROT_EXEC" constant: "#ifdef sgi; #define PROT_EXEC PROT_EXECUTE; #endif"
- In the *mvControl.h* file in the *include* subdirectory the definition of "m" was changed to 'm'
- In the *msTool.c* file in the *mains/tools/maxsp* subdirectory the variable const was changed to _const_
- In the *Mdepends.bsd* file in the *dc* subdirectory the "$(FUNCS): $(INCS)" line was commented out
- In the *setecho.c*, *clrecho.c*, *setcbreak.c*, and *clrbreak.c* files in the *parser* subdirectory the initializing declaration of "static struct termio tty = 0;" was changed to the simple declaration of " static struct termio tty"
- In the *roiCalc.c* file in the *mains/roicalc* subdirectory an "#ifdef sgi" was added for the System dependent constant variables that matched the "#ifdef sun"

```
/* entry allowing VME space to be mapped in
   specifically for the DigiColor and ROIStore boards */
#if IP5
        { VME_A24SSIZE, PHYS_TO_K1(VME_A24SBASE), },
        { VME_A32NPSIZE, PHYS_TO_K1(VME_A32NPBASE), },
#endif
```

Figure 6.4: Code added to kernel to allow it to map in VME space

```
if ((int)(roi_mem_ptr =
          mmap((caddr_t) 0, 0x80000, PROT_READ|PROT_WRITE,
               MAP_SHARED, fd,
               0xc80000+PHYS_TO_K1(VME_A24SBASE))) == -1) {
    perror("unable to perform mmap for ROIStore on /dev/mmem");
    exit(1);}
if ((int)(dc_mem_ptr =
           mmap((caddr_t) 0, 0x80000, PROT_READ|PROT_WRITE,
           MAP_SHARED, fd,
           0xe00000+0x40000+PHYS_TO_K1(VME_A24SBASE))) == -1) {
    perror("unable to perform mmap for DigiColor on /dev/mmem");
    exit(1);}
```

Figure 6.5: Code establishing pointers to VME space for direct access of boards

These changes were sufficient to compile all the code and successfully run the code for the DigiColor and ROIStore boards in “POLLED” mode.

6.2.2 Kernel Modifications

The Maxware code was originally ported to the IRIX 4.0.3 operating system. This version of the operating system did not contain any VME drivers so a method had to be found that would allow access to the DigiColor and ROIStore boards on the VME bus. There was a *mem* file in the *uts/mips/master.d* subdirectory that defined an array of device addresses mappable by the */dev/mmem*. The addresses must be kernel virtual addresses, not physical addresses. We treated the DigiColor and ROIStore boards as devices and specifically indicated in this file that it was valid for the kernel to map in this space - it just happened to be in VME space. The addition to the *mem* file shown in Figure 6.4 allows the boards to be *mmapped* with the code in Figure 6.5.

6.2.3 Digitization and Access

As can be seen in Figure 6.3 we used a color camera that was connected to a DigiColor board capable of digitizing an image into several different signals. The DigiColor board can produce a composite signal, a monotone signal, a RGB signal, and others. Our goal was to choose a background, objects, and signal that allowed for easy object detection; we were not attempting to solve a vision recognition problem. We did however, want to design a realistic problem, and decided not to place a high intensity point source of light on the objects. Instead, by choosing an appropriate background we could use a threshold to determine if a particular pixel was part of an object. As mentioned the sheep are constructed using off-the-shelf Lego pieces. While we had eventually planned to design a cover or "wool" for the sheep, we wanted to be able to also detect the standard Lego objects. For each of the possible output signals the DigiColor can produce, we took a histogram of the pixel intensities. We wanted to determine empirically the signal that produced the sharpest and furthest spread peaks for the background versus the object. The best results were obtained by using the green digitized signal produced by the DigiColor in RGB mode and treating it as a monochrome signal. Although a slightly sharper distinction could be made by adding the red and blue signals, this would have required additional cycles in transferring the digitized image between the DigiColor and ROIStore boards.

We determined that the thresholding can work under significantly different lighting conditions assuming that the program is appropriately initialized. Initialization is accomplished by obtaining a histogram of the pixel intensity values for a particular lighting condition and choosing a threshold value. The first threshold value we tried was the midway between the peaks of the background and objects. While this provided a reasonable value, through experimentation we discovered a value eighty to ninety percent nearer to the edge of the object peak provided for less noise around the perimeter of the objects.

There are several different methods of accessing the memory in the ROIStore and DigiColor boards. The Maxware primitives provide functions for accessing pixels individually or in blocks. Also by using pointers initialized as in Figure 6.5 we could circumvent several layers of maxware code and access the memory in the ROIStore directly over the VME bus. This technique yielded a factor of three increase in access time when accessing individual pixels. We further determined that in order for the block accessing maxware primitives to outperform our direct access method, it was necessary to access on the order of 10000 contiguous pixels. These pointers could also be used to access an image plane in the DigiColor memory that could be used to create overlays for images. This allowed us to display computer representations of the objects directly on the monitor displaying the actual image, and was useful both in debugging and observing the program.

6.3 Decomposition and Real-Time Control

In designing any large program, it is important to consider the software engineering aspects as well as the actual programming. An additional problem in implementing a real-world application is ensuring the individual modules are programmed so they run quick enough and are scheduled within the required time window. In this section we describe the module decomposition and the user-level scheduler we designed using IRIX real-time primitives to yield the desired behavior.

There were many hardware components used to implement the shepherding application. Figure 6.3 illustrates the connections between them. We used a PUMA robot arm to manipulate the sheep on the table. Control of the arm requires a software package called RCCL [LH92] that runs on a Sparc workstation. While we could have connected the vision processing boards to the VME bus on the same workstation and run all the vision processing, planning, display, and other functions on that same workstation, there was considerable motivation to implement all but the RCCL control on a more powerful SGI multiprocessor.

The decision to use the SGI was based in part on the realization that the vision processing portion required both intensive computation and extensive access to the ROIStore board on the VME bus. The SGI Challenge series with R4400 chips clocked at 100 MHZ and a 1.2 GByte bus, provided both fast processors and a fast bus. In addition to the increased power of the SGI, as mentioned in the introduction, we were interested in using the shepherding application to study and verify the issues involved in designing parallel real-world applications. Therefore, the vision processing boards were placed on the SGI. The goal was to place all the code possible on the SGI requiring as little as possible of the Sparcstation. All the vision processing code, the planning code (including where to send the robot arm), and the display code, was implemented on the SGI. Since the RCCL software needed to run on the Sun workstation, a method of communication between the Sun and SGI was required. The SGI performed the majority of the computation and simply sent desired robot arm coordinates to the Sun. This placed as little of a burden on the Sun as possible. All the Sun workstation needed to do was to perform a transformation between the image coordinates sent by the SGI and the robot world space, move the arm, and send back confirmation of success or failure.

The task running on the Sun workstation was straightforward: an infinite loop was set up to wait for a command from the SGI, perform a robot move, and send confirmation. On the SGI however there were many software modules that needed to be meshed. The module that coordinates and schedules the processes on the SGI is our user-level scheduler. It was designed in order to try new techniques and interactions between Ephor and the scheduler both rapidly and without kernel modification. After describing our user-level scheduler and the salient aspects of Ephor, we provide a description of the different modules comprising the shepherding application. The vision and manipulation modules receive more detailed treatment in later sections and thus are only briefly described here. Since the shepherding application ran on a multiprocessor, each module was given an individual processor. While highly cpu intensive modules (the vision processing and planning modules) could be parallelized as in our simulator, we found the real-world bottleneck was in the robot and communication, thus each module was assigned only one processor.

6.3.1 User-Level Scheduler

The user-level scheduler allowed Ephor to control the placement and timing of the tasks. In turn Ephor provides the user with a clean interface allowing easy specification of when and how frequently to run a particular task. In addition Ephor interacts with the user-level scheduler providing dynamic task selection, parallel process control, and more mechanisms for the SPARTA programmer.

It is necessary to have cooperation between the tasks and the user-level scheduler in order to simulate a real scheduler. The code for this functionality is placed in a header defined by a macro. Providing this macro removes responsibility from the user for providing the cooperation between Ephor and the user-level scheduler. All each task needs to do is to place a BEGIN_PROC (see Figure 6.6) statement at its beginning. BEGIN_PROC is a macro allowing the user-level scheduler to place this task on a specific processor at a given time. Combined with the END_PROC (see Figure 6.7) it allows for very precise timing.

The BEGIN_PROC macro sets each task into an infinite loop with a *blockproc* statement at the beginning followed by the code for the work. *Blockproc* causes a particular process to block until an *unblockproc* command is issued. The processes should be thought of as light-weight threads as they share address space and differ only in necessities such as the program counter, stack space, etc. After initialization a set of processes are created, one for each of the tasks. They have each executed the *blockproc* command appearing in the macro header. When the user-level scheduler needs to run a particular task it unblocks the (light-weight) process associated with that task.

Additional code in the BEGIN_PROC macro allows the user-level scheduler to also have control over the processor the task runs on. Each task (via code in the macro) executes a *sysmp(MUSTRUN, proc[my_id])* command. The task executing this command runs on the *proc* processor specified by the user-level scheduler. The array *proc* is set by the user-level scheduler for each task and can be changed dynamically. Another concern was getting accurate timings for the different tasks for scheduling purposes. The clock provided by the SGI only gave ten millisecond granularity for standard processes. While millisecond granularity was available to higher priority tasks, they are non-preemptable and we wanted the ability to allow the user-level scheduler to block tasks. An extremely high granularity clock was established on a distinct processor by using a shared variable and continually incrementing it. This provided 121 nanosecond resolution and was accurate to under one percent. All tasks were timed by checking this value in the BEGIN_PROC macro of the task and rechecking it in the END_PROC macro.

There are still more subtleties in ensuring the above mechanisms behave as expected. To guarantee the desired behavior, some additional IRIX real-time primitives were used. The processors were restricted (*sysmp(MP_RESTRICT,i)*) to running only the processes that had been assigned to them by the user-level scheduler. To ensure the tasks ran in the correct order the priorities associated with the processes were modified appropriately by a method similar to determining the processor to run on. The scheduler also had to track the execution times of the tasks and ensure that it placed only those that could run in the allocated time period.

The important aspect of Ephor and the user-level scheduler from the application programmer's perspective is the ease with which the user can specify timing constraints and priority concerns, and the increased performance achieved by Ephor's automatic mechanisms. The user calls initialization routines indicating whether that task is to be run periodically and its rate (e.g. for the object finding task) or whether it will be run in response to an environmental stimulus (the manipulation task).

6.3.2 Display Process

The *display* process was created primarily to provide a nice user interface allowing the programmer to view the computer's representation of the real world. This allowed both easier debugging and easier development. The interface also allows the user to control some of the actions of the application, e.g., whether to track objects to find them or scan the whole image to find them.

The *display* process starts by opening an X window with a portion of the window for displaying sheep positions and another portion containing action buttons. When blobs appear in the real-world scene, the *display* process obtains their image positions and displays them in the window as a circle. The area of the circle matches the area of the blob as found by the vision processing task. As the blobs move, their locations in the X window also move.

The X window can be used to control the computer program. One of the buttons appearing in the X window is a scan button. This effectively pauses the vision processing algorithm, i.e., it no longer searches for blobs in the image. This allows the user to enter the image and move objects around. Also as part of this mode, the user can move a computer generated box around the monitor displaying the image. This is useful in performing the initial robot calibration, where it is necessary to correlate points in the image to points in the robot's space. Other buttons include a pair indicating whether the program is currently tracking images and performing a perimeter search (trackify option) or whether the computer is trying to find blobs in the entire image (blobify option). Other options include whether to perform a full scan of a blob or just an axis scan. There is also a debug button allowing information (e.g., variables) to be dumped to the screen. This can be used to actually find a bug in the program or just a quick method to print out transient information about the objects. The details of these options and tools, and the tradeoffs and usefulness of some of them will be discussed in later sections.

6.3.3 Communication Process

There is communication between the SGI program and the robot control program on the Sparcstation. Communication is set up with sockets allowing transmission of short messages containing target image points and the verification of a robot move. A separate process is used since socket communication is a bottleneck in the system. If the *communication* process were joined with the *planner* process, considerable planning time would be lost while waiting for message replys. By spawning off a separate *communication* process, the system is allowed to continue processing visual input and planning intercepts while waiting for communication to complete.

When the *communication* process is first started, it has the robot control program (running on the Sparcstation) perform all necessary initialization. After initialization, the *communication* process spins in a loop waiting until the planner has decided on a robot manipulation. When a target position and manipulator orientation for the robot has been planned, the *communication* process sends the target and orientation to the Sparcstation for processing, and waits for a reply verifying the arm movement. Upon confirmation, the *communication* process notifies the *planner* process, via a flag, that the motion is complete. It then waits for the next robot position to send.

6.3.4 Planner Process

Determining the next sheep to save can take up a considerable amount of time depending on the strategy employed. It is possible to imagine a broad spectrum of possibilities ranging from picking the first sheep found moving away from the center, to exploring all possibilities of the next n sheep to save (exponential). One of the mechanisms in Ephor allows a user to program many different possibilities and will automatically and dynamically select the most appropriate one at runtime. Unfortunately with the hardware in our laboratory this decision was relatively uninteresting because we could not physically get that many sheep on the table. (The puma arm we have is slow and has limited reach). However, this did bring up other tradeoffs based on the limitations of the robot arm, e.g., it had problems reaching one corner of the field so it was best to herd sheep away from it earlier than for the other corners. Another difficulty in the laboratory was the fact that the robot was extremely slow in comparison to the processors, allowing considerable computation for each manipulation move. We still did investigate the tradeoffs between the planner described above and one that exhaustively searched the entire space.

The real-world shepherder has two versions of the planner: the single sheep version and the multi-sheep version. Both are described in the following paragraphs.

The single sheep planner is designed to restrict the motion of a single sheep. When the sheep is spotted, an intercept point is found on the field boundary, the robot positions itself at the intercept point in the elevation plane (a plane far enough above the field to allow free movement of the robot arm) with the manipulator oriented to catch it, and then the manipulator is lowered to the object plane (the plane of the field). The intercept point is calculated by using our simulator to project the sheep movements into the future. Until the sheep reaches the intercept point, the position and orientation of the manipulator are corrected to compensate for slight changes. This is accomplished by by continuously monitoring sheep and robot position as they approach each other. When the sheep reaches the manipulator, it is reoriented toward the center of the field. Finally, the manipulator is removed from the path of the sheep and placed in a location such that the arm will not obscure any of the field (it is placed in the lower right corner). This *planner* process is currently very simple but is sufficient to contain one sheep. It allowed us to verify that the other hardware and software components functioned as required.

To contain more sheep, a more complicated planner is required. First, a sheep that needs to be saved is chosen according to the following criteria: 1) it will leave the scene sooner than any other sheep, and 2) it is not already headed toward the center of the scene. If no sheep fits the conditions (no sheep need to be saved yet), then the arm moves to the lower right corner of the scene to prevent sheep from being obscured.

Second, an intercept is calculated according to arm position and velocity, sheep position and velocity, and a delay long enough to allow the arm to move from the elevation plane into the object plane. If the intercept is outside the field, or if it intercepts the expected location of a second sheep, then the first sheep is ignored and a new sheep is considered for rescue.

Third, after the intercept has been accepted, the arm is moved to that location in the elevation plane. Once the move has been made, the intercept is rechecked to ensure no sheep are directly under the arm (it is possible that before the arm moves no sheep other than the target will be at the intercept, but during the move two sheep could collide directing a sheep to the intercept point). After the second intercept verification has been made, the arm is lowered into the object plane (directly over the target sheep), the sheep is redirected toward the center of the scene, and the arm moves into the elevation plane.

Finally, this process starts all over again by either going to save another sheep, or moving to the corner of the scene if no sheep currently needs to be saved.

6.3.5 The Vision Process

The *vision* process performs a search on the image data for groups or blobs of pixels indicating the positions of the sheep; this is known as the blobify routine. The *vision* process is run at a specified regular interval and controlled by the *scheduler* process. The *vision* process is run 20 times a second, so the blobify routine must be fast enough to complete once during every interval. As described later, we needed a high scan frequency to meet the constraints placed on the real-time vision processing portion by the shepherding application.

6.3.6 The Manipulation Process

The *manipulation* process resides on the Sparcstation controlling the robot arm. It is sent an arm position and manipulator orientation specified in image coordinates. First, the *manipulation* process converts the image coordinates into robot world coordinates (this calculation is described later). Second, it makes sure the desired speed is possible. This is not straightforward since speed is specified in Cartesian coordinates (pixels per second and radians per second), while the arm speed is constrained by the six independent joint velocities. The check is done by performing inverse kinematics to calculate new joint positions. Then each joint is checked to make sure none exceeds its maximum velocities. If a joint velocity will be out of bounds, then the overall speed is reduced to allow the arm to perform the movement. Once the arm speed is verified, the move is made and the *manipulation* process sends a message back to the SGI verifying the new position of the arm.

6.4 Vision

Detecting and tracking moving objects in the real world places a set of timing constraints on the vision processing portion of an application that differ from standard vision processing. Additionally, the shepherding application requires very accurate velocity. To satisfy the real-time constraints, the vision processing algorithm must execute both quickly and be predictable from execution to execution to allow the scheduler to calculate a suitable interval. For example, if the processing time spent on finding objects varied significantly, then that portion of the algorithm may not finish in the expected interval causing stale data to be used and thus invalidating the velocity prediction portion of the algorithm. The calculated velocity is used in the shepherding application to predict where the object will be many (perhaps one hundred) steps in the future. Thus, in addition to tracking the current position of the objects, it is very important in the shepherding application that an accurate velocity be provided to the $\alpha - \beta$ filter since any error in velocity will be multiplied many times as the future position is predicted. To obtain an accurate velocity we need to maintain a high resolution image, i.e., subsampling yields less accurate positions and thus less accurate velocities. In the shepherding application there are many objects (sheep) that could be in the field simultaneously. It was necessary to track multiple objects and to associate blobs in the present image with objects from the previous images (we use *blob* for a incoherent group of pixels and *object* for a processed coherent group). It was also possible that some of the objects would be obscured for variable lengths of time. It was therefore necessary to develop an algorithm capable of handling obscured objects. The vision portion of the shepherding application needed to be able to track multiple objects rapidly, run with small time variation, and be able to produce very accurate velocity prediction.

We present an overview of the vision algorithm here and describe each phase in more detail in the upcoming sections. The first stage in the algorithm is to determine where the blobs are. Blobs represent probable objects. Once these have been obtained it is necessary to associate the blobs with objects. In many cases this is a fairly simple operation since using a high scan rate prevents the objects from shifting significantly between snapshots. However, it is possible that due to object collisions or an object being obscured this initial match is not successful. To handle these situations, obscured objects are given projected positions and "tracked" while "new" objects are remapped onto the old expected objects. After a complete pairing has been accomplished the object positions are used to calculated a preliminary velocity. This velocity however is noisy due to the very noisy movements of the sheep and the noise associated with the camera. The velocities are passed through a double $\alpha - \beta$ filter with distinct parameters and different sampling rates for each filter. The output of the second filter is taken to be the "true" velocity and the data structure associated with that object is updated. This velocity can then be used to predict future positions of that object.

6.4.1 Blobify

The first stage in the vision algorithm is to group the pixels above the threshold value into blobs. This blobification process is simplified because we used a solid background with a pixel value lower than that of the objects. The actual objects (sheep) we used are displayed in Figures 6.8 and 6.9. As mentioned earlier, the object finding algorithm worked on the uncovered sheep. There were, however, two strong motivating factors for providing "wool" for the sheep. The bare Lego sheep provided for non-elastic collisions that tend to clump sheep throughout the field making for an uninteresting shepherding problem. The jagged edges and wires produced an additional source of noise (the number of detectable pixels had a much higher variance). Further, the appearance of bare Lego sheep is extremely susceptible to variations in lighting conditions due to the shiny surfaces of the pieces. The wooly sheep provided a more consistent size under similar lighting conditions in different portions of the image and were also more consistent across different lighting conditions. Our motivation in placing the wool on the sheep was to allow us (when implementing our real-world application) to focus on the real-time tracking and real-time association problems rather than the object detection problem. Should the need occur to track natural, the simple thresholding function would be replaced with a suitable object detection function.

Blobs are searched for in a 512×482 image stored in the ROIStore memory as a one-dimensional array after being digitized by the DigiColor. Blobifying the image can easily be the most time consuming stage since each pixel must be referenced over the VME bus. It was at this stage that we had to utilize intelligent algorithms that differed from standard vision processing ones. We needed to blobify the entire image quickly and at high resolution. Our goal was to be able to process the entire image at a rate between twenty and thirty Hertz. To reference every pixel in the ROIStore required .8 sec. Even performing a bcopy from VME space to main memory of the image required .3 sec, plus the time to access them from memory. Clearly it is not possible to examine every pixel, or even a significant 'percentage of them, and meet a twenty Hertz constraint. We therefore developed a two phase examination of the pixels.

Conceptually the algorithm is broken into two phases. In the first phase we perform a very sparse subsampled search. We use information about the size of the blobs and try to make the search in this phase as sparse as possible without missing a sheep. The object of this phase was to produce plausible locations for blobs. A sparse two-dimensional shadow array (a sample one is shown on the left side of Figure 6.10) of the actual image is kept in main memory and upon identification of a possible object a cell is marked. A true value in the shadow array indicates a possible blob for expansion in the second phase(the X's in Figure 6.10). The X's in Figure 6.10 indicate cells in the sparse array that would contain a pixel above threshold. Notice that an X occurs in the sparse array only if there is a corresponding pixel in the image that would be in that cell.

The second phase involves performing a high resolution search on the pixels in the actual image corresponding to those marked in the shadow array in the first phase. The goal is to determine the centroid of the blob in both the x and y axes. This will be used as its position to calculate velocity and then will be passed through a double $\alpha - \beta$ filter. There are several possible methods to perform a search. One algorithm is based on the simplifying assumption that the blobs are close to circular. The first step in this algorithm is to scan up and down from the original point. The algorithm performs a scan in each direction until it determines, by pixel intensity value that it is at the end of the blob. This obtains a first preliminary vertical line as shown in circle **1** of Figure 6.11. A center point is determined for the vertical line and a horizontal scan is performed in each direction, again until the end of the blob is found. This operation finds the horizontal diameter of the blob (see circle **2** of Figure 6.11). As a check that the blob was shaped close to expected (a circle) a third vertical scan can be performed as in circle **3**. If the ends of the blob are approximately the same distance away from the horizontal diameter, then the center of mass is the point at which the two diameters cross. It is possible that in performing the check the algorithm determines the blob was not circular.

In many cases, as was true for our shepherding, the circle assumption could not be made. Even though the spots on the sheep's wool were circular (Figure 6.8), they could become obscured. While becoming obscured, the spot loses its circular shape. We performed experiments and determined that even with reasonably slow moving sheep and a slightly faster robot arm, this effect could throw the estimated velocity off by up to a factor of five (making making simulated movement into the future impossible). This is because the center of mass changes much more rapidly as a blob is becoming obscured. It may well also be the case the image being blobified does not consist of circular blobs. To handle non-circular blobs, a recursive search of all the pixels in the blob is performed about the point corresponding to the one in the sparse array. Since speed was of primary concern, rather than using function recursion we implemented a stack of pixels and performed the entire recursive search with one function call. A step is made in each of four directions until the end of the blob is encountered. A cumulative total of x and y values and number of points was kept. After the entire blob has been explored the x and y sum is divided by the number of points to determine the x and y centroids.

In reality, as an optimization, the phases are combined. As soon as a pixel is found in the sparse scan, we pause phase one and perform a high resolution scan about this point. After the blob is explored in the high resolution image the corresponding cells are marked as invalid (do not expand) in the sparse shadow matrix as in Figure 6.10. This eliminates having to check to make sure duplicate objects are not produced. It also eliminates extra references to pixels in the ROIStore, i.e., if the stages were not combined there would be many more forward examinations of each possible hit in the sparse array. The result is a high resolution blobify routine that runs very quickly due to a reduced search space and a fast interactive version of a detailed, recursive, search function. When the blobification process completes, an array of blob positions denoted by x and y centroids has been updated.

Instead of searching for blobs in the entire image, an optimization is possible if we assume objects can only enter the field from the perimeter (sheep can't fly). We will call this algorithm *trackify* since it involves looking for object based on where they were in the last image. The center of each object is used as the initial point in the sparse matrix and a high resolution scan is performed using it as the origin. Then a general scan, as previously described, is performed around the perimeter. For this to be valid the objects cannot move more than the distance equal to their radius between frames, otherwise the initial point (the center of the object from the last image) will not be part of the blob. This is a reasonable assumption. In our case sheep moved two or three pixels a second and their spots were about sixteen pixels in diameter. We scanned at 20Hz. Thus, we had about 80 images before the center of the blob would no longer be a point anywhere in the blob.

Driving the hardware at this rate is challenging. The DigiColor can digitize half (every other line) the image at 60Hz. Thus a complete new full image image is available at 30Hz. Trying to synchronize the algorithm with the scan rate (by checking appropriate flags) and the rest of the code may not have been doable at 20Hz. Instead, we perform a continuous scan disregarding where the DigiColor is in writing the image. This could produce a situation where garbled data was being used if we happened to be reading in the area the DigiColor was writing. To avoid this difficulty we scan backwards in ROIStore memory while the DigiColor writes the digitized image forwards. In this way, the possibility of accessing pixels being written is minimized and the area of potential overlap is reduced to a few pixels.

The difference between performing a trackify and blobify operation can be significant. As illustrated in Figure 6.12, there can be a significant difference between performing these two operations. The sampling rate indicates the ratio between the sparse matrix and the actual image. For example, a sampling rate of eight indicates every eighth pixel was examined in the first phase of the algorithm. The numbers in the table were gathered performing a high resolution recursive search with six blobs in the image each with of diameter of approximately sixteen pixels. Performing the full high resolution scan versus making the circular assumption adds only about .3msecs per blob or 1.8msecs for the numbers in the table. Since this number is small we always perform the full search when blobifying in the shepherding application.

6.4.2 Associating Blobs with Objects

The next step is to associate blobs with objects (sheep). Blobs are associated with objects for several reasons. The planner requires velocity estimates as well as position estimates. A blob is only a snapshot in one image. Storing consecutive positions in an object structure allows continuous velocity estimates to be obtained by using a filter. This is important because at any time the planning process may need to know the position and velocity of a particular sheep. There are, however, difficulties that arise with attempting to provide continuous positions to the filter. When the robot arm moves over a sheep the sheep becomes obscured and a blob is no longer reported for that object. If the blob was being stored as an object we can continue to estimate where that object will be. Objects also admit the ability to disambiguate two blobs with velocity vectors as shown in Figure 6.13. As sheep 1 moves down past sheep 2 the blobifier will form the blobs in a different order. This figure illustrates that it is not possible to simply feed the same blob center positions to a filter or any other permanent data structure; it is important that an intelligent mapping between blobs and objects occur. In order to update sheep positions while obscured, calculate velocities, and associate a sheep with a single, nonchanging set of values, blobs need to be associated with specific objects.

The task of associating blobs to objects is non-trivial. The process is broken down into four stages executed in the following order: 1) associate visible blobs to known objects, 2) map unassociated blobs to recently unobscured objects, 3) estimate the positions of obscured objects, and 4) map unassociated blobs to new objects.

1) Associate Visible Blobs to Known Objects

This step is straightforward. Since we use a high scan rate, objects move less than a pixel from snapshot to snapshot. A pointer is kept to the blob a particular object was associated with last time. If the blob's new position is consistent with the last position and velocity estimate of the object, then the association is kept. If there is no blob whose position is consistent with the object, then it is assumed that the sheep has become obscured and the object is marked as such. Also, if the object's new position is outside of the field (the sheep escaped the confined area), then the object is marked as dead and removed from the list of objects.

2) Map Unassociated Blobs to Recently Unobscured Objects

Once blobs have been associated with objects, there might be some unassociated blobs left over. This can occur for one of two reasons: either a blob has recently been obscured and has just move out from under the robot arm, or an entirely new sheep has appeared on the table. A search is made through all the objects that have been marked as obscured to determine if one of those might match the blob in question. This match is determined by using the estimated positions and velocities of obscured objects. If the blobs current position is within a predetermined percent of where the object was expected to be, an association is established. A positive association may not be made if while the blob was obscured it was manipulated back toward the center, or if it collided with another sheep. The allowed percent is increased (effectively widening the search) until a positive match is found. Our double pass filters allow for the velocity estimate to track the object's new heading quickly, so even if it is the case that the object has been turned around, the velocity estimate will soon (within two to three snapshots, or about a tenth of a second) correctly reflect the object's true velocity.

3) Estimate Positions of Obscured Objects

When a sheep becomes obscured, it is necessary to estimate its position according to its last measured position and velocity. This is done simply by multiplying the amount of time the object has been obscured with its last estimated velocity. This is another reason why it is important to have very accurate and quickly determinable velocities. The position estimate kept when obscured is used when trying to reassociate a blob detected on the field. It is also used by the planner to determine the next sheep to save. It is possible that an obscured sheep needs to be reoriented so that it won't escape the field. If the estimated position of an object is ever outside of the field, it is marked as dead and the sheep is removed from the object list.

4) Map Unassociated Blobs to New Objects

By this last stage in object to blob association, if there are any unassociated blobs, they are assumed to be new and are mapped to a new object. Hence, a new object is added to the list, the position information is added, and the filtering process begins. In our shepherding application, the "trackify" assumption is that objects are placed only on the perimeter. The "blobify" algorithm allows new objects to appear anywhere in the scene. For instance, a sheep entering the field may slide under the robot arm and be obscured until it has already crossed a portion of the field. While careful placement would avoid this difficulty, we designed the blobify algorithm to handle such a scenario.

6.4.3 Filtering

Once blobs have been associated with objects, another operation is needed before a final velocity is ready for the planner. It is necessary to filter the data.

Figure 6.14 shows velocity data for a sheep standing perfectly still. With no noise, the velocity would be a stable zero. Since the object is stationary, this figure represents the sensor noise introduced by the camera and digitizer. The camera noise represents a significant portion of the total noise. The Lego sheep introduce plant noise since they do not move at constant speed; the exact quantity however is difficult to capture. An examination of Figure 6.15 (this figure shows the velocity of a sheep moving in one direction, a 180 degree rotation, followed by velocity in the opposite direction) shows periodic noise patterns indicating that the sheep are adding their own noise on top of the measurement noise produced by the camera. Successive figures in this section represent combined plant and sensor noise.

To reduce the effects of plant and measurement noise, an $\alpha - \beta$ filter was used to incorporate new data into old [BSF88]. The sheep occasionally change direction, even by as much as 180 degrees (when being manipulated back towards the center), or less when deflecting off other objects. As stated, the planner needed very consistent velocity estimates. The requirements of the filtered velocity were that it be smooth and representative of the averaged true velocity and respond quickly to changes in the object's motion. An $\alpha - \beta$ filter has two parameters that can be set. These represent how much confidence should be extended to the data. If the filter is set to have a high degree of confidence in the data, the filtered output will very closely resemble the input data, and any change in direction will quickly be reflected in the filtered output. However, if the plant noise is large, then the filtered output will still be quite noisy. If the parameters are set reflecting little confidence in the data, then the filtered output is smooth but responds slowly to changes in direction. We were thus faced with a dilemma when choosing the parameters. In fact, several experiments confirmed our suspicion that there would be no appropriate choice of parameters for a single $\alpha - \beta$ filter.

To solve this problem, we ran the noisy input data through a double $\alpha - \beta$ filter with different parameters for each filter. The first filter's parameters (see table 6.16) are set assuming small plant noise. This is so the filtered output will quickly respond to changes in directions of the objects. The original data appears in Figure 6.15. The output from the first filter appears in Figure 6.17 and although it is still fairly noisy, it closely tracks changes in direction of the sheep. As mentioned, the eventual output needed to be smooth for accurate prediction by the planner. To achieve this, the output of the first filter was used as input to a second filter. The second filter assumed a higher plant noise (for greater smoothing) and a much lower measurement noise (the data had already been filtered so it shouldn't be as noisy as the original data). Additionally, we subsampled the first filter data, taking every third point, to produce an even smoother curve. The output of the second filter appears in Figure 6.18. The output of the second filter fulfills the requirements: it is quite smooth and responds quickly to changes in direction of the sheep.

To see how well the output of the second filter mimics the original input data, as well as the intermediate stage, a graph overlaying the data presented in Figures 6.15, 6.17, and 6.18 is presented in Figure 6.19. The velocity from the second filter fulfills the two requirements of having a small delay in adjusting velocities when there is a change in direction and of having a smooth profile.

To gauge the accuracy of the filtered velocity estimate, we ran several experiments in which a sheep was introduced into the field and the filter was allowed to establish a velocity estimate. A pseudo-planner asked for the sheep's velocity and predicted the sheep's position n seconds into the future. The pseudo-planner waited for the n seconds and asked for the position of the sheep. We recorded the difference of the predicted position of the sheep with the actual position. We performed this experiment with n taking on values of 1, 2, 4, 8, and 16. For each n, we ran six experiments and took the mean of the absolute values of the difference between the predicted and actual position. The graph in Figure 6.20 represents the results. The y axis is in pixels. The approximate velocity of the sheep was .8 pixels per second. Remember, the diameter of the sheep is about 16 pixels. The graph indicates very accurate prediction and the accuracy is not significantly affected by increasing time. This is true for two reasons: the plant noise (the velocity) is periodic so when integrated over time is predictable, and there is a large amount of error arising from sensor noise (the measurement of the positions of the sheep). These experiments were run with a 20HZ scan rate; 16 seconds represents multiplying the estimated velocity by 320 to obtain the predicted position. These results clearly indicate the estimate velocity from the double filter is very accurate.

6.5 Manipulation

One of the most difficult aspects of real-time manipulation is that the robot arm is extremely slow and unpredictable [1]. The program that controls the Puma robotic arm runs on a Sparcstation using the RCCL robot control software. As we previously described, this program is sent requests from the SGI where the vision processing and planning occur. The task of the program running on the Sparcstation is to move the arm to a specified (x, y) point with a given orientation θ. It is not straightforward since the SGI specifies a target point in two-dimensional image co-ordinates. The manipulation program must use a transformation matrix to convert between the requested image coordinate specified by the GSI and the robot's three-dimensional world coordinate frame.

[1]The arm does not arrive at the exact coordinate specified and sometimes takes the much longer way (of two possible rotations sometimes the internal RCCL path finder chooses the longer one to prevent getting too close to an end position) to get to a destination

To simplify the problem, the robot is required to work in only two given z-planes: the object plane where it can manipulate the sheep, and the elevation plane where the arm can move freely without bumping the sheep. Due to the specification of z-planes, the transformation requires a 2×3 matrix.

To create the transformation matrix, three sample points must be taken to set up a correspondence between image and world points and mark the object plane, and a fourth point is taken to specify the elevation plane. Once the sample points have been taken and the transformation matrix is created, the operation in equation (6.1) must be made to convert image points into world points.

$$\begin{bmatrix} T_{11} & T_{12} & T_{13} \\ T_{21} & T_{22} & T_{23} \end{bmatrix} \begin{bmatrix} x_I \\ y_I \\ 1 \end{bmatrix} = \begin{bmatrix} x_W \\ y_W \end{bmatrix} \tag{6.1}$$

In the initialization phase, the elements of the 2×3 matrix shown in equation (6.1) must be found. It is clear that that can be done by getting three world points and their associated image points and solving the set of three equations to find the three unknowns (three unknowns per row of the transformation matrix). The object plane is specified by the user in the first image to world point correspondent.

It is also simple to find the world co-ordinate orientation of the gripper given the image co-ordinate orientation. A vector from the origin of the image can be easily computed given an orientation (in radians). Then, using the transformation matrix, the corresponding world vector can be found. With this information it is possible to orient the manipulator in the world.

The final aspect of robot control is the speed of the arm. Fortunately, RCCL provides a function to set the time it takes to get to a target point. Given the current location of the arm and the destination point (in image co-ordinates), we can use this function to find the travel time of the arm. Unfortunately though, this is sometimes in error and an allowance for this possibility was required. The planner also has access to the arm speed constant allowing it to estimate how long a requested move will take.

6.6 Example Application Conclusions

The real-world shepherding application runs in our laboratory. The setup of the field, robot arm, and sheep can be seen in Figure 6.21. The robot arm can keep one sheep on the field indefinitely, indicating that it can track and manipulate. It also performs well when two sheep are present, however mechanical difficulties sometime interfere. If the two sheep collide and stick the robot doesn't need to do anything for the sheep to remain in the field. If the two sheep stick and move together the manipulator we have designed for the end of the robot arm does not allow it to turn the sheep back towards the center. Both of these events are unlikely with two sheep on the field but increase as we place more sheep on the field. The other difficulty with greater numbers of sheep is the relatively limited extent the robot can reach. This caused the size of the field to be quite small. The size of the field is approximately three feet by four feet, and each sheep is nine inches long and three and half inches wide. We have been able to confine four sheep on the field for a limited amount of time (about two saves each). The most limiting factor is the speed of the robot arm.

We have used the basic shepherding platform to create graduate student class projects. The implementation was robust enough to allow students to use it as a base and work with additional interesting variations. One group, *no helicopters*, was not allowed to have an overhead camera or to survey the field from any height above several inches. This precluded the possibility of obtaining a global view of the world. A second group, *clouds*, had to handle multiple clouds (objects resting above the field that obscured the sheep) throughout the field. This made for frequently obscured sheep and placed additional constraints upon quick acquisition and accurate tracking of the objects. As a final variation, *wolves*, we used a second robot arm to simulate a wolf entering the field. The original robot arm was equipped with a laser gun and had to "kill" the encroaching wolf by aiming at and hitting a sensor target placed on the second robot arm controlling the wolf.

We have described the implementation of shepherding, a real-world application combining vision, manipulation, and planning. This application has been successfully implemented in our hardware lab and can confine approximately four sheep in a three foot by four foot field. This project was part of a larger project of investigating the design issues of implementing support for parallel real-world applications. The implementation has allowed us to address areas where simulation was inadequate. However, simulation allows us to address situations the real-world application does not. A better understanding of support and design and principles for SPARTAS comes from a combination of both techniques.

```
#define ephor_begin_proc(TASK,ID) \
  long long ftime1,ftime2;\
  int *cpu_times, *cpu_index;\
  int *sd_times, *sd_index;\
  int hltrt = hl_task[TASK].run_technique;\
  cpu_times = hl_task[TASK].technique[hltrt].cpu_times;\
  cpu_index = &(hl_task[TASK].technique[hltrt].cpu_index);\
  sd_times = hl_task[TASK].technique[hltrt].sd_times;\
  sd_index = &(hl_task[TASK].technique[hltrt].sd_index);\
  while(start_signal == 0);  /* delay so proc id stablizes */\
  if (verbose) printf("%s id %d\n",hl_task[TASK].name,ID);\
  hl_task[TASK].proc = FIRST_FREE_PROCESSOR;\
  if (sysmp(MP_MUSTRUN, hl_task[TASK].proc) < 0)\
        ephor_error("error: failed to assign processor);\
  hl_task[TASK].migrate = 0;\
  setblockproccnt(ID,0);\
  schedctl(NDPRI,0,NDPHIMAX);\
  cpu_on[hl_task[TASK].proc] = 0;\
  while (1) {\
    blockproc(ID);\
    if (hl_task[TASK].pri_add)\
      schedctl(NDPRI,0,NDPHIMAX+hl_task[TASK].pri_add);\
    if (hl_task[TASK].migrate) {\
      if (sysmp(MP_MUSTRUN, hl_task[TASK].proc) < 0)\
        ephor_error("error: failed to assign processor); \
      hl_task[TASK].migrate = 0;\
    }\
    ftime1 = *fine_timer;
```

Figure 6.6: C code for the ephor_begin_proc macro

```
#define ephor_end_proc(TASK) \
    hl_task[TASK].technique[hltrt].cpu_marked = 1;\
    ftime2 = *fine_timer;\
    (*cpu_index)++;\
    (*sd_index)++;\
    cpu_times[(*cpu_index)%3] = TICS2USECS((int)(ftime2-ftime1));\
    sd_times[(*sd_index)%sd_window] = \
          TICS2USECS((int)(ftime2-ftime1));\
    cpu_on[hl_task[TASK].proc] = 0;\
}
```

Figure 6.7: C code for the ephor_end_proc macro

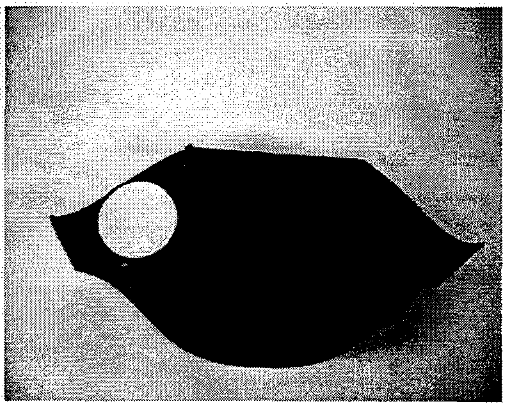
Figure 6.8: A covered Lego sheep.

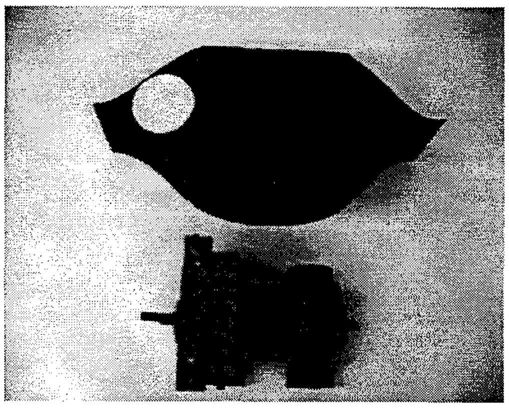
Figure 6.9: A sheep and cover.

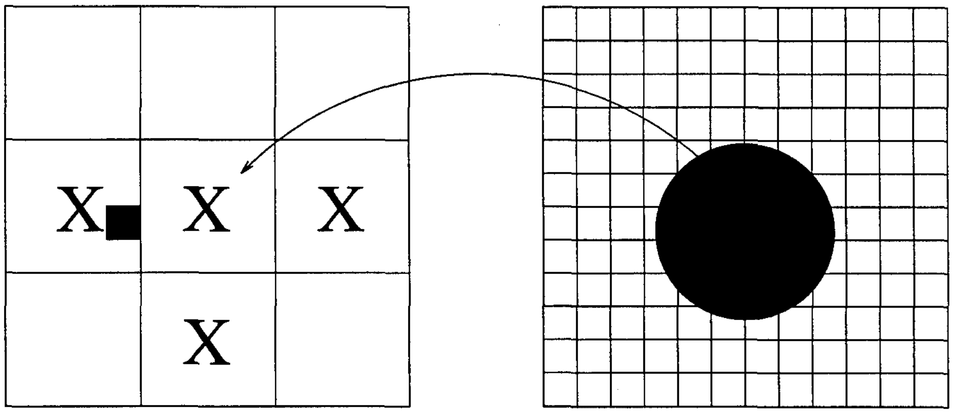

Figure 6.10: The actual image (on the right) and its shadow array (on the left)

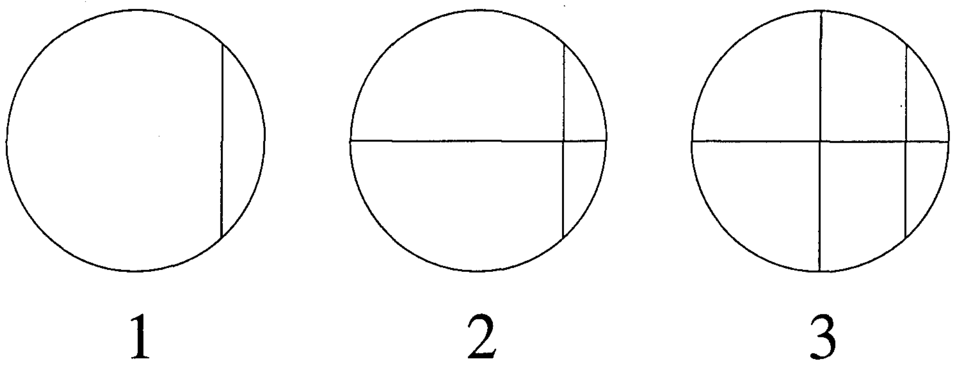

Figure 6.11: The lines created by the quick circle method

Sampling rate	blobify	trackify
4 pixels	26 ms	18 ms
8 pixels	17 ms	8 ms
10 pixels	12 ms	4 ms
12 pixels	9 ms	3 ms

Figure 6.12: Comparing blobify to trackify

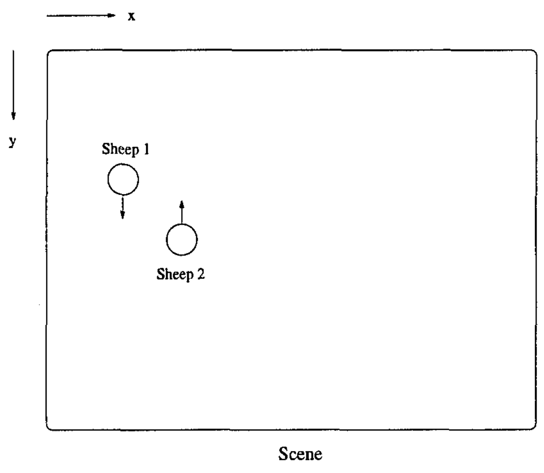

Figure 6.13: A sample scene with two sheep and their velocity vectors.

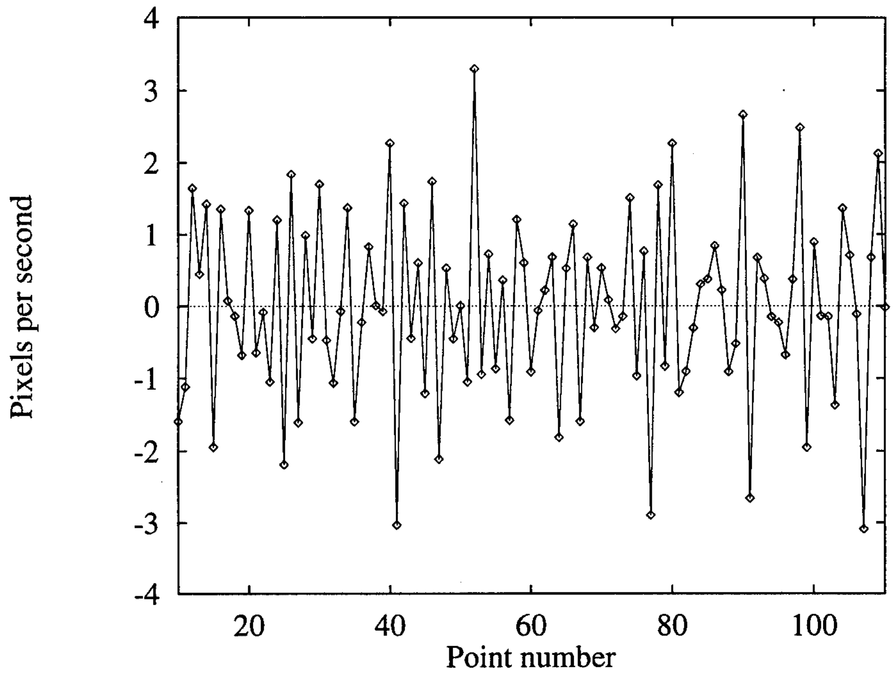

Figure 6.14: Velocity for a stationary sheep

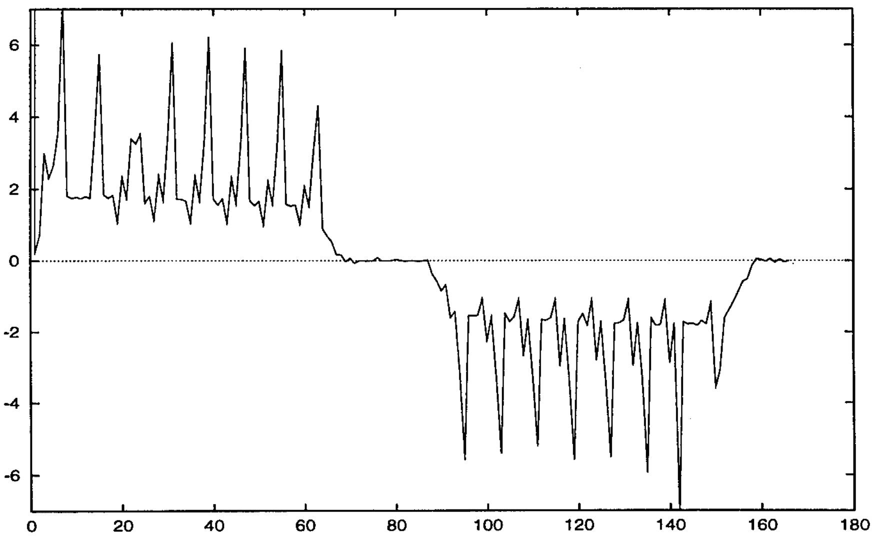

Figure 6.15: The actual measured velocity of the toy sheep where the x-axis is in twentieths of a second and the y-axis is pixels per second.

Plant noise filter 1	0.01
Measurement noise filter 1	0.1
Plant noise filter 2	0.05
Measurement noise filter 2	0.01

Figure 6.16: The parameters used in both of the $\alpha - \beta$ filters.

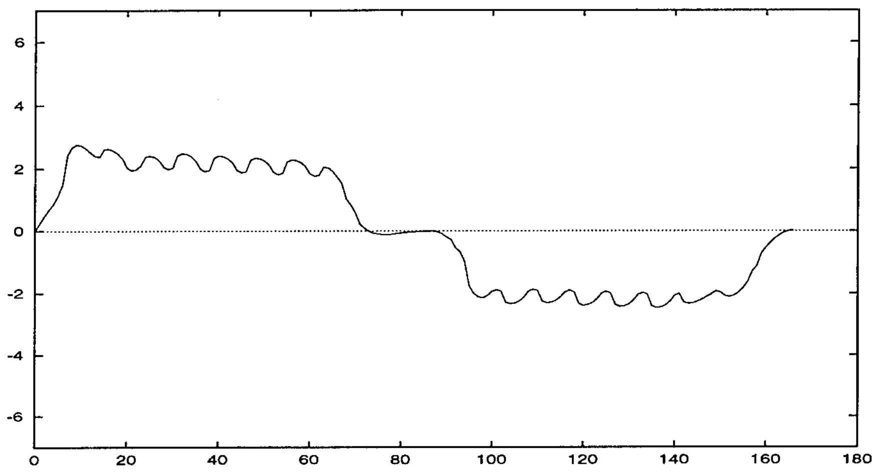

Figure 6.17: The output of the first $\alpha - \beta$ filter.

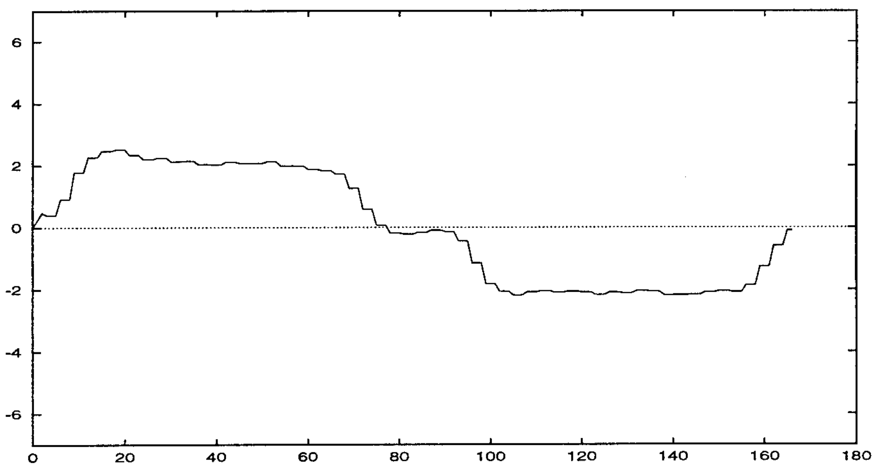

Figure 6.18: The output of the second $\alpha - \beta$ filter.

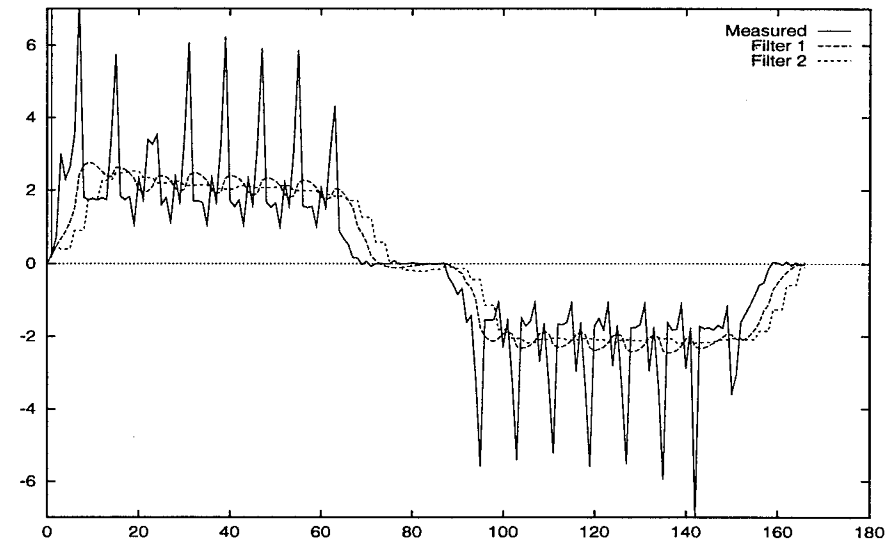

Figure 6.19: The measured, first filter, and second filter velocities.

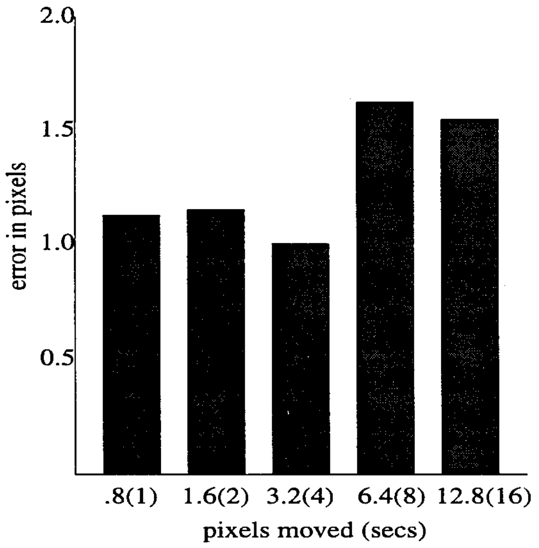

Figure 6.20: Error in predicting a sheep position.

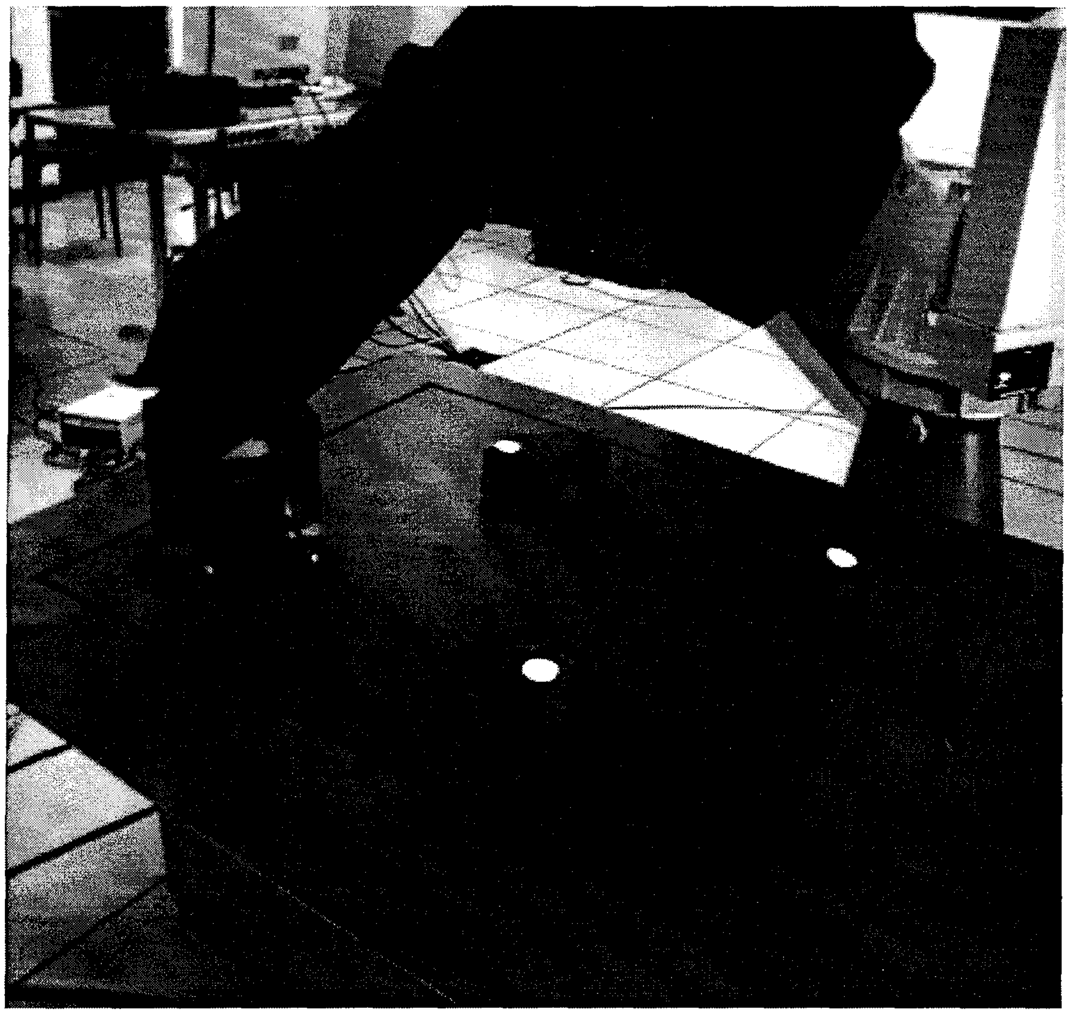
Figure 6.21: Shepherding setup

7 Conclusions

Parallel applications face challenges when handling unexpected events such as a context switch or a new environmental stimulus. Uncertainty makes programs more difficult to design and causes large performance degradation if not handled properly. Our approach to achieving high performance in the face of uncertain events has been to widen the kernel-application interface allowing the two to cooperate. We applied our approach to the design of synchronization primitives on multiprogrammed multiprocessors and to allow parallel real-time applications to adapt to a varying environment. Our expanded interface and cooperation allowed the application code to be simpler and to perform better.

7.1 Contributions

Part of designing a successful SPARTA involves partitioning the responsibilities to facilitate clean interaction between the application and the system. In Chapter 2 we provided

- an application independent structure and methodology for incorporating useful mechanisms into the runtime environment.

In designing Ephor we focused on mechanisms targeted at real-world environments. One observation made about the real world is that it can be modeled by continuous functions and often does not change dramatically over short periods of time (e.g.., relatively smooth number of cars into and out of a city block). Based on this, in Chapter 4 we presented

- a set of scheduling policies designed for SPARTA environments.

The underlying goal of these policies was to be adaptive and respond to a changing environment. This is a key element to designing well-performing SPARTAs. This observation led us, in Chapter 5, to designing a validating the usefulness of

- a dynamic technique selection mechanism

that allows the runtime (Ephor) to choose dynamically from a set of possible techniques to executing a particular task. In the same vein we also described

- a suite of mechanisms suitable for SPARTAs: dynamic parallel process control, de-scheduling of all tasks linked to a goal, access functions for sharing information, optional partitioning of soft and hard tasks, overdemand detection and recovery, and early termination of tasks.

Given this effective base and our experience with the real-world shepherding application (Chapter 6), in Chapter 2 we presented

- a set recommendations to SPARTA programmers.

Uncertainty can arise both from the environment and from the internal system state. Our dynamic technique selection allowed Ephor to adapt to varying internal state by choosing the appropriate mechanism when a task needed to execute. Another aspect to internal state is the multiprogramming level of the system, which can have a severe effect on the performance of parallel applications employing synchronization. To address these issues, in Chapter 3 we described techniques for:

- a preemption-safe ticket lock and scheduler-conscious queue lock, both of which provide FIFO servicing of requests and scale well to large machines
- a fair, scalable, scheduler-conscious reader-writer lock
- a scheduler-conscious barrier for small machines (in which a centralized data structure does not suffer from undue contention, and in which processes can migrate between processors)
- a scheduler-conscious barrier for large machines that are partitioned among applications, and on which processes migrate only when repartitioning.

Throughout our work, the emphasis has been on using the widened kernel - application interface to facilitate the cooperation between the application and the kernel to allow for easier implementation of better performing mechanisms.

7.2 Future Directions

Further extensions to the kernel-user interface may allow even greater performance gains to be achieved. Several groups have examined exporting access to the hardware and kernel responsibility. The SPIN project [BSP+95] allows the kernel to run user defined function providing protection through Modula 3, a strictly typed language. Wahbe et al. [WLAG93] use sandboxing, a technique allowing fault isolation within a single address space. The Exokernel [EKJ95] exposes resources to the user and provides protection via hardware mechanisms. These projects all recognize the fact the in the past operating systems have hid potentially critical implementation decisions from the user. We believe the ability to expose these decisions to the user and allow an alternative implementation will be crucial to achieving high performance in current and future scalable machines. However, we also feel it is vital to not force the user to perform additional work. Instead it should be an option should they so chose, with the underlying runtime by default acting on the application's behalf.

There are several extensions to current operating systems we believe may be valuable. Such extensions might include allowing the kernel to run user-supplied functions in response to particular kernel events, or choosing the partition size based on the application's characteristics. We believe that as large-scale multiprocessors become more common they will inevitably be multiprogrammed, and the importance of exchanging information across the kernel-user boundary will increase.

Clusters of SMPs and scalable shared memory multiprocessors will be important architectures in the near and medium term future. An interesting question is "how do we schedule applications on these machines?" For example, if a barrier application has 16 processes running on 16 processors, removing 1 processor may have the same effect as removing 8, thus it may be much better (for overall performance) to give the other 7 to another application. Also in a clustered environment the choice of those 8 may be important. Here again, we believe if the OS scheduler has information about the application, better decisions can be made. Another interesting issue for the operating system is whether the scheduler should itself be distributed. This would allow greater tolerance to single cluster failure but has added complexity in overall scheduler design.

Some machines do not provide the fetch_and_Φ operations needed by many synchronization algorithms. Thus synchronization mechanisms that rely on those will have difficulty, yet synchronization is crucially important to many parallel applications. Those instructions could be emulated using simpler atomic primitives, but whether the additional overhead is justified remains an open question. Perhaps a two-tiered mechanism similar to the one we used in our scalable barrier algorithms will need to be employed.

In Chapter 3 we discussed ticket locks and concluded that designing a scheduler-conscious version was not possible with simple sharing of information. However, if the kernel knew more about the synchronization mechanisms, the lock would be implementable. This hints at the potential for allowing the kernel to run user functions to achieve better performance. There are many interesting and challenging issues once such activity is considered.

Ephor uses program structure to dynamically select from a set of possible implementations. This idea opens up a whole set of possibilities. For example, if a program could specify ahead of time its resource allocation needs, the operating system could pre-allocate the requisite resources. In real-time systems this is sometimes done to avoid the variability of process creation and other similar functions. However, we believe there is potential for it to be used in other realms as well.

We presented many different useful mechanisms in Ephor, but only explored a few of their combinations. Some of the methods we employed for resolving conflicts between competing goals were very simple. A more in-depth understanding of the application, or again the ability to allow Ephor to run user-level functions, would perhaps provide improved performance when many tasks are competing for interelated resources.

We believe there is tremendous potential to be gained by extending and improving the exchange of information between the application layer and the system layer. There are many issues involved including protection, security, and reliability in widening this interface. This thesis demonstrated the benefits to be gained in a few specific arenas and we would like to see this work continue and be expanded into other related domains.

Bibliography

[ABLL92] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *TOCS*, 10(1):53–79, February 1992.

[AJ89] N. S. Arenstorf and H. Jordan. Comparing barrier algorithms. In *Parallel Computing*, volume 12, pages pp. 157–170, 1989.

[And90] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEETPDS*, 1(1):pp. 6–16, January 1990.

[Axe86] T. S. Axelrod. Effects of synchronization barriers on multiprocessor performance. In *Parallel Computing*, volume 3, pages pp. 129–140, 1986.

[BB92] D.H. Ballard and C. M. Brown. Principles of animate vision. *cvgip*, 56(1):3–21, July 1992.

[BDW86] J. Blazewicz, M. Drabowski, and J. Weglarz. Scheduling multiprocessor tasks to minimize schedule length. *IEEE Transaction on Computers*, pages 389–393, May 1986.

[Bla90] D. L. Black. Scheduling support for concurrency and parallelism in the mach operatin system. *IEEEC*, 23(5):35–43, May 1990.

[Bro87] Rodney A. Brooks. Intelligence without representation. In *Proceedings of the Workshop on the Foundations of Artificial Intelligence*, 1987.

[Bro89] Rodney A. Brooks. A robot that walks: Emergent behavior from a carefully evolved network. *Neural Computation*, 1(2):253–262, 1989.

[BS91] Thomas E. Bihari and Karsten Schwan. Dynamic adaptation of real-time software. *ACM Transactions on Computer Systems*, 9(2):143–174, May 1991.

[BSF88] Y. Bar-Shalom and T. E. Fortman. *Tracking and Data Association.* Academic Press, 1988.

[BSP$^+$95] Brian N Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, Marc Fiuczynski, David Becker, Susan Eggers, and Craig Chambers. Extensibility, safety and performance in the spin operating system. *Proceddings of the 15th SOSP*, December 1995.

[BT94] Christopher M. Brown and Demetri Terzopoulos. *Real-Time Computer Vision*, chapter Chapter 9: Hybrid problems need hybrid solutions? Tracking and controlling toy cars, pages 209–230. Cambridge University Press, 1994.

[CDD$^+$91] Mark Crovella, Prakash Das, Cesary Dubnicki, Thomas LeBlanc, and Evangelos Markatos. Multiprogramming on multiprocessors. In *PROC of the Third SPDP*, pages pp. 590–597, December 1991.

[CG93] C. Connolly and R. Grupen. On the applications of harmonic functions to robotics. *J. Robotic Systems*, 10(7):931–946, 1993.

[CG95] C. Connolly and R. Grupen. Nonholonomic path planning using harmonic functions. In *Int. Conf. Robotics and Automation*, 1995.

[CJ91] Sarah E. Chodrow and Farnam Jahanian. Run-time monitoring of real-time systems. *IEEE Real-Time Systems Symposium*, pages 74–83, 1991.

[Cok91] Ronald S. Cok. *Parallel Programs for the Transputer.* Prentice Hall, 1991.

[Cra93] Travis S. Craig. Queueing spin lock algorithms to support timing predictability. In *Real-Time Systems Symposium*, pages 148–157, Raleigh-Durham NC, December 1-3 1993.

[DHK$^+$88] M. Daily, J. Harris, D. Keirsey, K. Olin, D. Payton, K. Reiser, J. Rosenblatt, D. Tsneg, and V. Wong. Autonomous cross–country navigation with the alv. In *IEEE 1988 Internation Conference on Robotics and Automation*, pages 718--726, jan 1988.

[DJ86] Marc D. Donner and David H. Jameson. A real–time juggling robot. *Real Time Systems Symposium*, pages 249–256, 1986.

[DPAB95] Robert Davis, Sasikumar Punnekat, Neil Audslet, and Alan Burns. Flexible scheduling for adaptable real-time systems. In *Real-Time Technology and Applications*, pages 230–239, Chiacgo IL, May 1995. IEEE.

[Dur90] Edmund H. Durfee. Towards intelligent real-time cooperative systems. In James Hendler, editor, *Planning in Uncertain, Unpredictable, of Changing Environments: 1990 AAAI Spring Symposium*, 1990.

[EHRLR80] L. D. Erman, F. Hayes-Roth, V. R. Lesser, and D. R. Reddy. The hearsay-ii speech-understanding system: Integrating knowledge to resolve uncertainty. *Computing Surveys*, 12(2):213–253, June 1980.

[EKJ95] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. *Proceddings of the 15th SOSP*, December 1995.

[EL75] L. D. Erman and V. R. Lesser. A multi-level organization for problem solving using many diverse cooperating sources of kowledge. In *Proceedings of the 4th IJCAI*, pages 438–490, September 1975.

[ELS88] J. Edler, A J. Lipkis, and A E. Schonberg. Process management for highly parallel unix systems. *PROC of the USENIX Workshop on Unix and Supercomputers*, September 26-27 1988.

[Fod85] Jerry A. Fodor. Precis of the modularity of mind. *The Behavioral and Brain Sciences*, 8(1):1–5, 1985.

[Gan96] Roger F. Gans. Following behavior using moving potential fields. Technical Report 603, Department of Computer Science, University of Rochester, Rochester, NY 14627, January 1996 1996.

[GCSB95] R. Grupen, C. Connolly, K. Souccar, and W. Burleson. Toward a path co-processor for automated vehicle control. In *Unknown Conf.*, 1995.

[GD92] Melinda T. Gervasio and Gerald F. DeJong. Completable scheduling: An integrated approach to planning and scheduling. In *AAAI Spring Symposium on Practical Approaches to Scheduling and Planning*, pages 122–126, Stanford University, March 1992.

[GS89] Prabha Gopinath and Karsten Schwan. Chaos: Why one cannot have only an operating system for real-time applications. *Operating Systems Review*, 23(3):106–125, July 1989.

[GT90] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEEC*, 23(6):pp. 60–69, June 1990.

[Hal90] John Hallam. *CARS: An Experiment in Real-Time Intelligent Control*, 1990.

[Her91] M. Herlihy. Wait-free synchronization. *TOPLAS*, 13(1):pp. 124–149, January 1991.

[Her93] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *TOPLAS*, 15(5):pp. 745–770, November 1993.

[HFM88] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *IJPP*, 17(1):pp. 1–17, 1988.

[HH89] V. P. Holmes and D. L. Harris. A designer's perspective of the hawk multiprocessor operating system kernel. *Operating Systems Review*, 23(3):158–172, July 1989.

[HH92] Boudewijn Hoogeboom and Wolfgang A. Halang. *Real-Time Systems Engineering and Applications*, chapter 2: The Concept of Time in the Specification of real-Time Systems, pages 11–40. Kluwer Academic Publishers, 1992.

[HHP92] Khosrow Hadavi, Wen-Ling Hsu, and Michael Pinedo. Adaptive planning for applications with dynamic objectives. In *AAAI Spring Symposium on Practical Approaches to Scheduling and Planning*, pages 30–31, Stanford University, March 1992.

[HS89] Dieter Haban and Kang G. Shin. Application of real-time monitoring to scheduling tasks with random execution times. *IEEE Real-Time Systems Symposium*, pages 172–181, 1989.

[JLGS90] D. V. James, A. T. Laundrie, S. Gjessing, and G. S. Sohi. Scalable coherent interface. *IEEEC*, 23(6):pp. 74–77, June 1990.

[KJA$^+$93] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, and B.-H. Lim. Integrating message-passing and shared-memory: Early experience. In *PROC of the Fourth PPOPP*, San Diego, CA, May 20-22 1993.

[KLMO91] A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *PROC of the Thirteenth SOSP*, pages pp. 41–55, Pacific Grove, CA, October 13-16 1991.

[KP93] Jon G. Kuhl and Tiannis E. Papelis. A real-time software architecture for an operator-in-the-loop simulator. In *Proceedings of The Workshop on Parallel and Distributed Real-Time Systems*, pages 117–126, Newport Beach, California, April 13-15, 1993.

[KSU93] O. Krieger, M. Stumm, and R. Unrau. A fair fast scalable reader-writer lock. In *PROC of the 1993 ICPP*, St. Charles, IL, August 16-20 1993.

[KW93] Leonidas Kontothanassis and Robert W. Wisniewski. Using scheduler information to achieve optimal barrier synchronization performance. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 64–72, San Diego, CA, May 19-22 1993.

[KWS94] Leonidas I. Kontothanassis, Robert W. Wisniewski, and Michael L. Scott. Scheduler conscious synchronization. Technical Report 550, Department of Computer Science, University of Rochester, Rochester, NY, December 1994.

[LA94] Beng-Hong Lim and Anant Agarwal. Reactive synchronization algorithms for multiprocessors. In *PROC of the Sixth ASPLOS*, pages pp. 25–35, San Jose, CA, October 5-7 1994.

[Lee90] C. A. Lee. Barrier synchronization over multistage interconnection networks. In *PROC of the Second SPDP*, pages pp. 130–133, Dallas, TX, December 1990.

[LH92] John Lloyd and Vincent Hayward. *RCCL - RCI System Overview*. McGill Research Centre for Intelligent Machines, McGill University, Montreal, Quebec, Canada, July 1992.

[LHM$^+$84] Bruce Lindsay, Laura M. Haas, C Mohan, Paul F. Wilms, and Robert A. Yost. Computation and communication in r*: A distributed database manager. *TOCS*, 2(1):pp. 24–28, February 1984.

[LL73] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real–time environment. *JACM*, pages 46–61, February 1973.

[LLG$^+$92] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The stanford dash multiprocessor. *IEEEC*, 25(3):pp. 63–79, March 1992.

[LLS$^+$91] Jan Liu, Kwei-Jay Lin, Wei-Kuan Shih, Albert Yu, Jen-Yao Chung, and Wei Zhao. Algorithms for scheduling imprecise computations. *IEEE Computer*, 24(5):58–69, May 1991.

[Lub89] B. Lubachevsky. Synchronization barrier and related tools for shared memory parallel programming. In *PROC of the 1989 ICPP*, pages pp. II:175–179, August 1989.

[LV90] S. T. Leutenegger and M. K. Vernon. Performance of multiprogrammed multiprocessor scheduling algorithms. In *PROC of the 1990 MMCS*, Boulder, CO, May 22-25 1990.

[MBL$^+$92] B. Marsh, C. Brown, T. LeBlanc, M. Scott, T. Becker, P. Das, J. Karlsson, and C. Quiroz. Operating system support for animate vision. *Journal of Parallel and Distributed Computing*, 15(2):103–117, June 1992.

[MCS91a] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *TOCS*, 9(1):pp. 21–65, February 1991.

[MCS91b] J. M. Mellor-Crummey and M. L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *PROC of the Third PPOPP*, pages pp. 106–113, Williamsburg, VA, April 21-2 1991.

[MCS91c] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. In *PROC of the Fourth ASPLOS*, pages pp. 269–278, Santa Clara, CA, April 8-11 1991.

[ML91] Evangelos P. Markatos and Thomas J. LeBlanc. Load balancing versus locality management in shared-memory multiprocessors. Technical Report TR 399, Department of Computer Science, University of Rochester, Rochester, NY, September 1991.

[MLH94] Peter Magnussen, Anders Landin, and Erik Hagersten. Queue locks on cache coherent multiprocessors. In *PROC of the Eighth IPPS*, Cancun, Mexico, April 26-29 1994.

[MNS87] A P. Moller-Nielsen and A J. Staunstrup. Problem-heap: A paradigm for multiprocessor algorithms. *Parallel Computing*, 4:64–74, 1987.

[MSLM91] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First–class user–level threads. In *Proceedings of the Thirteenth SOSP*, pages 110–121, Pacific Grove, CA, October 1991.

[NP91] Vivek Nirkhe and William Pugh. A partial evaluator for the maruti hard real-time system. *IEEE Real-Time Systems Symposium*, pages 64–73, 1991.

[NWD93] M. Noakes, D. Wallach, and W. Dally. The j-machine multicomputer: An architectural evaluation. In *PROC of the Twentieth ISCA*, San Diego, CA, May 16-19 1993.

[OQC91] E Oliveira, L Qiegang, and R Camacho. Controlling cooperative experts in a real-time system. *Software Engineering for Real-Time Systems*, pages 176–181, September 1991.

[Ous82] John K. Ousterhout. Scheduling techniques for concurrent systems. In *PROC of the Third ICDCS*, pages pp. 22–30, October 1982.

[PABS91] C. J. Paul, Anurag Acharya, Bryan Black, and Jay K. Strosnider. Reducing problem-solving variance to improve predictability. *Communications of the ACM*, 34(8):80–93, August 1991.

[RK89] E. Rimon and D. Koditschek. The construction of analytic diffeomorphisms for exact robot navigation on star worlds. In *Proc. Int. Conf. on Robotics and Automation*, pages 21–26, 1989.

[RSS90] Krithi Ramamritham, John A. Stankovic, and Pering-Fei Shiah. Efficient scheduling algorithms for real–time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):184–194, April 1990.

[SG90] J. W. Stamos and D. K. Gifford. Remote evaluation. *TOPLAS*, 12(4):pp. 537–565, October 1990.

[SG92] U. M. Schwuttke and L. Gasser. Real-time metareasoning with dynamic trade-off evaluation. In *Proceedings of the Tenth National Conference on Artificial Intelligence AAAI 1992*, pages 500–506, 1992.

[SGB87] K. Schwan, P. Gopinath, and W. Bo. Chaos–kernel support for objects in the real-time domain. *IEEE Transactions on Computers*, 36(8):904–916, August 1987.

[Sha87] Steven A. Shafer. The navlab new generation vision system: 1987 plan for integration. project description, February 1987.

[SLM90] M. L. Scott, Thomas J. LeBlanc, and B. D. Marsh. Multi-model parallel programming in psyche. In *PROC of the Second PPOPP*, pages pp. 70–78, Seattle, WA, March 14-16 1990.

[SMC94] M. L. Scott and J. M. Mellor-Crummey. Fast, contention-free combining tree barriers. *IJPP*, 22(4):pp. 449–481, August 1994.

[SP94] Jay K. Strosnider and C. J. Paul. A structured view of real-time problem solving. *AI Magazine*, 15(2):45–66, summer 1994.

[SR87] J. Stankovic and K. Ramamritham. The design of the spring kernel. *Proceedings of the Real–Time Systems Symposium*, pages 146–157, December 1987.

[SS89] Igor Steinberg and Marvin Solomon. Searching game trees in parallel. Technical Report 877, Department of Computer Science, University of Wisconsin, Madison, WI, September 1989.

[SS91] Tobiah E. Smith and Dorothy E. Setliff. Towards and automatic synthesis for real-time software. *IEEE Real-Time Systems Symposium*, pages 34–42, 1991.

[Sta92] John A. Stankovic. Real-time operating systems: What's wrong with today's systems and research issues. *Real-Time Systems Newsletter*, 8(1):1–9, 1992.

[SWG92] J. P. Singh, W.-D. Weber, and A. Gupta. Splash: Stanford parallel applications for shared-memory. *CAN*, 20(1):pp. 5–44, March 1992.

[TB91] L. Tarassenko and A. Blake. Analogue computation of collision-free paths. In *Proc. Int. Conf. on Robotics and Automation*, pages 540–545, 1991.

[TC88] R.H. Thomas and A W. Crowther. The uniform system: An approach to runtime support for large scale shared memory parallel processors. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 245–254, St. Charles IL, August 1988.

[TFC90] Jeffrey J. P. Tsai, Kwang-Ya Fang, and Horng-Yuam Chen. A non-invasive architecture to monitor real-time distributed systems. *Computer*, 23(3):11–23, March 1990.

[TG89] A. Tucker and A. Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *PROC of the Twelfth SOSP*, pages pp. 159–166, Litchfield Park, AZ, December 3-6 1989.

[TL94] Sandra R. Thuel and John P. Lehoczky. Algorithms for scheduling hard aperiodic tasks in fixed-priority systems using slack stealing. In *IEEE Real-Time Systems Symposium*, pages 22–33, San Juan, Puerto Rico, December 1994.

[TS94] Hiroaki Takada and Ken Sakamura. Predictable spin lock algorithms with preemption. In *Real-Time Operating Systems and Software*, pages 2–6, Seattle WA, May 18-19 1994. IEEE Computer Society Press.

[vCGS92] T. vonEicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. In *PROC of the Nineteenth ISCA*, pages pp. 256–266, Gold Coast, Australia, May 1992.

[vKW94] Peter von Kaenel and Robert W. Wisniewski. Real-world shepherding - combining vision, manipulation, and planning in real time. Technical Report 530, Department of Computer Science, University of Rochester, Rochester, NY, August 1994.

[WB93] Robert W. Wisniewski and Christopher M. Brown. Ephor, a run-time environment for parallel intelligent applications. In *Proceedings of The IEEE Workshop on Parallel and Distributed Real-Time Systems*, pages 51–60, Newport Beach, California, April 13-15, 1993.

[WB94] Robert W. Wisniewski and Christopher M. Brown. An argument for a runtime layer in sparta design. In *Proceedings of The 11th IEEE Workshop on Real-Time Operating Systems and Software*, pages 91–95, Seattle Wahington, May 18-19 1994.

[WB95] Robert W. Wisniewski and Christopher M Brown. Adaptable planner primitives for real-world ai applications. In *Proceedings of the International Joint Conference on Artificial Intelligence*, volume 1, pages pp 64–71, Montreal Canada, August 20-25 1995.

[WB96] Robert W. Wisniewski and Christopher M. Brown. Scheduling mechanisms for spartas. Technical Report 604, Department of Computer Science, University of Rochester, Rochester, NY 14627, January 1996.

[WHR90] Richard Washington and Barbara Hayes-Roth. Abstraction planning in real-time. In James Hendler, editor, *Planning in Uncertain, Unpredictable, of Changing Environments: 1990 AAAI Spring Symposium*, 1990.

[WKS94] Robert. W. Wisniewski, L. Kontothanassis, and M. L. Scott. Scalable spin locks for multiprogrammed systems. In *Proceedings of the Eighth International Parallel Processing Symposium*, pages 583–589, Cancun Mexico, April 26-29 1994.

[WKS95] Robert W. Wisniewski, Leonidas I. Kontothanassis, , and Michael L. Scott. High performance synchronization algorithms for multiprogrammed multiprocessors. In *Proceedings of The Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages pp. 199–206, Sanata Barabara California, July 19-21 1995.

[WLAG93] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. *Proceddings of the 14th SOSP*, pages 203–216, December 1993.

[YA93] H.-H. Yang and J. H. Anderson. Fast, scalable synchronization with minimal hardware support (extended abstract). In *PROC of the Twelfth PODC*, August 15-18 1993.

[YTL87] P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEETC*, C-36(4):pp. 388–395, April 1987.

[ZLE88] J. Zahorjan, E. D. Lazowska, and D. L. Eager. Spinning versus blocking in parallel systems with uncertainty. In *PROC of the INTL SYMP on Performance of Distributed and Parallel Systems*, December 1988.

[ZLE91] J. Zahorjan, E. D. Lazowska, and D. L. Eager. The effect of scheduling discipline on spin overhead in shared memory parallel systems. *IEEETPDS*, 2(2):pp. 180–198, April 1991.

[ZM90] J. Zahorjan and C. McCann. Processor scheduling in shared memory multiprocessors. In *PROC of the 1990 MMCS*, pages pp. 214–225, Boulder, CO, May 22-25 1990.

[ZRS87] A W. Zhao, A K. Ramamritham, and J.A. Stankovic. Preemptive scheduling under time and resource constraints. *IEEE Transactions on Computers*, C–36(8):949–960, August 1987.